

High-Bandwidth/Low-Latency Temporary Storage
for Supercomputers

By

John Alan Swensen

A.B. (University of California) 1979

M.S. (University of California) 1982

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved:.....*Yuh N. Pao*.....17 November 87
Chairman Date
.....*[Signature]*.....19 November 1987
.....*George J. Berlekamp*.....18 November 87

.....



**High-Bandwidth/Low-Latency Temporary Storage
for Supercomputers**

Copyright © 1987

John Alan Swensen



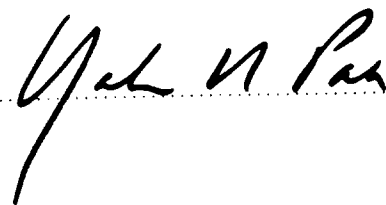
High-Bandwidth/Low-Latency Temporary Storage for Supercomputers

John Alan Swensen

Abstract

The traditional use of memory and a symmetrical set of registers for storage of temporary results of scientific programs requires more execution time, hardware, and instruction-stream bandwidth than necessary. Novel register organizations that can be easily integrated into traditional supercomputer architectures can reduce all of these requirements.

Execution speed can be more than doubled by storing temporary results in an asymmetrical set of general-purpose registers or an asymmetrical set of vector registers, instead of in memory and a small register-set. Faster access and a hardware cost one fourth that of traditional vector registers can be had by using a vector register that incorporates a pipelined, random-access-memory chip. If a large enough set of registers is used, the need to store temporary results in memory and then reload them for later use can be eliminated; this saves both instruction-stream bandwidth and execution time.

A handwritten signature in cursive script, appearing to read "John A. Swensen", is written over a horizontal dotted line.



Dedication

To Cindy, and to someone yet unnamed and unborn.

Acknowledgements

This work was supported in part by Lawrence Livermore National Laboratories, LLNL Contract 4695505. I gratefully acknowledge the continued enthusiastic support of the scientists at Livermore, in particular George Michael and John Ranelletti.

I have been fortunate to have Yale Patt as my advisor for several years. I have learned a great deal from him about computer architecture, computer science, life in general, and the importance of proper tool usage. He is perhaps unique among the professors at Berkeley because he consistently does what he believes is best for the student, even though it may be to his personal disadvantage. He has given me the freedom to pursue the areas I believe to be important, he has shielded me from the nefarious politics in the department, and he has not been above carrying boxes across the building to free up office space.

Alvin Despain first introduced me to Kuck's text, Earle Latches, and serious computer design. Always encouraging, he has often challenged my approaches to problems and has forced me investigate areas I would not have thought to consider. The net effect has been to improve the quality of my work, for which I am grateful.

In my first meeting with Elwyn Berlekamp, I discovered that we share an appreciation for the CDC 6600. He later shared with me his insights into program-tuning for supercomputers, and introduced me to my first thesis topic. After lightning struck and I discovered that my dissertation had been written by someone else, I returned to my notes on Elwyn's insights into hardware design; I eventually got from there to my current thesis topic. Thus, Elwyn Berlekamp should be thanked for giving me two thesis topics, both of which resulted in degrees.

Velvel Kahan taught me everything I know about floating-point arithmetic, much of what I know about military history, and some of what I know about automobiles. He shares with Yale a rare quality of truly caring about the well-being of the student. I am flattered that he has given me his time so freely, and that he has attempted to educate me about things of beauty.

Some of engineering ability is learned, and some is probably genetic; in any case, Alan and Betty Swensen deserve much of the credit for what I have accomplished. Alan taught me the basics of structural engineering, and tolerated my early experiments in material-strength and in failure-mode analysis. He taught me about electricity and electronics, and unwittingly tolerated my later experiments in the current-handling abilities of milliammeters. He also taught me how much fun hard work can be. Betty taught me about the beauty and balance that can be present in anything, from a musical phrase to the shape of a boat. She was always eager to listen to what I had to say, and she was rarely, if ever, judgemental. She also taught me how much work can be accomplished by staying up all night.

Together, they provided a loving and supportive environment that I could always retreat to. Because of them, I had the courage to attempt goals that seemed impossible; some of these I did achieve.

Table of Contents

Chapter 1: Introduction	1
1.1. Problem Description	1
1.2. Thesis Statement	3
1.3. Approach	4
1.4. Research Contributions	4
1.5. Terminology Conventions	6
1.6. Organization of this Dissertation	7
Chapter 2: Survey of Previous Work	10
2.1. Optimization and Scheduling	10
2.1.1. Optimization	11
2.1.2. Scheduling	12
2.1.2.1. Static Scheduling	12
2.1.2.2. Dynamic Scheduling	13
2.2. Application Characteristics and Program Structure	14
2.3. Datapath-Design Issues	17
2.3.1. Pipelining	17
2.3.2. Arithmetic Unit Design	18
2.3.3. Decoding Delays	19
2.3.4. Components	19
2.4. Storage-Design Issues	20

2.5. Instruction Issuing and Instruction Formats	23
2.6. Other Factors	26
2.7. Past and Present Supercomputers	26
Chapter 3: Program Characteristics	29
3.1. Livermore Kernel Benchmark Descriptions	30
3.2. Dependency Graphs	31
3.2.1. Dependency-Graph Generation	32
3.3. Program Partitioning	33
3.3.1. Partitioning Example	35
3.3.2. Prediction of Execution Times from Partitions	37
3.3.3. Partition-Size Analysis	42
3.3.4. Memory-Reference Analysis	43
3.3.5. Temporary-Storage-Requirement Analysis	44
3.4. Execution Simulation	46
3.4.1. Dependence of Performance on Number of Registers	48
3.4.2. Fast-Register Requirements	58
3.5. Summary of Program Characteristics and Hardware Requirements	65
Chapter 4: Datapath Design	67
4.1. Signal-Propagation Delays	67
4.2. Data Routing	71
4.2.1. Basic Selection Delays	71
4.2.2. Unbalanced Selection Trees	74
4.2.3. Data-Routing Delays	76

4.3. Pipeline Design and Analysis	78
4.3.1. Clocking Constraints for Polarity-Hold Latches	80
4.3.2. Latency-Throughput Tradeoffs for Pipelines	82
4.3.2.1. Variable-Speed Pipelines	83
4.3.2.2. Tuning Pipeline Parameters to Workloads	87
4.3.2.3. Replicated Pipelines	90
4.3.3. Concessions to Slow Chip I/O	91
4.4. Summary of Datapath-Design Properties	92
Chapter 5: Temporary Storage Devices and Access Mechanisms	94
5.1. Close/Distant General-Purpose Registers	95
5.1.1. Logic Design	97
5.1.2. Close/Distant-Register Allocation	101
5.1.3. Instruction-Stream Bandwidth Considerations	102
5.2. Fast/Bulk Vector Registers	104
5.2.1. Advantages of Fast/Bulk Vector Registers	109
5.2.2. Vector-Access Extensions	111
5.2.2.1. Parallel Vector Access	111
5.2.2.2. Simultaneous Read and Write Access to a Vector Register	111
5.3. Pipelined Random-Access-Memory Chips	115
5.3.1. Memory-Access Delays	116
5.3.2. Limited Memory-Access Pipelining	117
5.3.3. Signal-Propagation Pipelining	119
5.3.4. PRAM Applications	122

Chapter 6: Analysis of Some Existing Supercomputers	125
6.1. Cray-1	125
6.1.1. Description	125
6.1.2. Analysis	126
6.1.3. Execution Rates	130
6.2. Cray X-MP	131
6.2.1. Description	131
6.2.2. Analysis	132
6.2.3. Execution Rates	134
6.3. Cray-2	134
6.3.1. Description	135
6.3.2. Analysis	136
6.3.3. Execution Rates	137
6.4. NEC SX-2	138
6.4.1. Description	138
6.4.2. Analysis	139
6.4.3. Execution Rates	141
6.5. HEP-1	142
6.5.1. Description	142
6.5.2. Analysis	143
6.5.3. Execution Rates	144
6.6. ETA-10	145
6.6.1. Description	145

6.6.2. Analysis	147
6.6.3. Execution Rates	148
Chapter 7: Architectural Integration	150
7.1. Cray-2 Architecture	150
7.2. LARC (Large Asymmetrical-Register Computer) Architecture	152
7.2.1. Hardware Description	152
7.2.2. Larc Instruction Formats	155
7.3. Discussion	158
7.4. Execution Rates	159
Chapter 8: Concluding Remarks	167
8.1. Summary of Results	167
8.1.1. Summary of Program Characteristics	167
8.1.2. Datapath Design	168
8.1.3. Temporary Storage Devices and Access Mechanisms	170
8.1.4. Architectural Integration	171
8.2. Discussion	171
8.3. Conclusions	173
8.4. Future Work	174
Appendix A: References	175
Appendix B: Livermore Kernel Descriptions	186
Appendix C: Execution Time Predictions and Simulations	190

List of Figures

Figure 1.1: Hypothetical Code Sequence for $W = X * Y + Z$	1
Figure 3.1: Dependency Graph for Livermore Kernel 6 with $n = 3$	35
Figure 3.2a: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 1.	49
Figure 3.2b: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 2.	50
Figure 3.2c: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 3.	50
Figure 3.2d: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 4.	51
Figure 3.2e: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 5.	51
Figure 3.2f: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 6.	52
Figure 3.2g: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 7.	52
Figure 3.2h: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 8.	53
Figure 3.2i: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 9.	53
Figure 3.2j: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 10.	54

Figure 3.2k: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 11.	54
Figure 3.2l: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 12.	55
Figure 3.2m: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 13.	55
Figure 3.2n: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 14.	56
Figure 4.1: Fanning a Datum Out from One Source to Eight Destinations, Using Gates with a Maximum Fan-Out of 2.	72
Figure 4.2: Multiplexing Eight Data Down to One, Using Gates with a Maximum Fan-In of 3.	72
Figure 4.3: 4:1 Multiplexer with Decoding Circuitry, Using Gates with a Maximum Fan-Out of 2 and a Maximum Fan-In of 3.	74
Figure 4.4: Unbalanced Fan-Out Tree, Fanning a Datum Out from One Source to Ten Destinations, Using Gates with a Maximum Fan-Out of 2.	75
Figure 4.5: Unbalanced Fan-In Tree, Fanning Data from Eight Sources In to One, Using Gates with a Maximum Fan-In of 2.	75
Figure 4.6: Earle Latch Implemented Using AND Gates and OR Gates.	79
Figure 4.7: Earle Latch that Computes the Exclusive-OR of X and Y	79
Figure 4.8: Polarity-Hold Latch Implemented Using AND Gates and OR Gates.	80
Figure 4.9: Radial Arrangement of Functional Units in a CPU, Minimizing Propaga- tion Distances to and from Registers and Functional Units.	82
Figure 4.10: Polarity-Hold Latch with Wire-Pad Switching-Circuit.	85

Figure 5.1: Organization of Close/Distant Registers.	96
Figure 5.2: Implementation of a Polarity Hold Latch that Performs Selection and Logical Combination.	99
Figure 5.3: Implementation of a Traditional 64-Element Vector Register.	105
Figure 5.3: Uncle George's Pontoon Boat.	105
Figure 5.4: Vector Register Access Timing for a Write and a Read of a 7-Element Vector.	106
Figure 5.5: Implementation of a Fast/Bulk Vector Register with 128 Elements.	107
Figure 5.6: Fast/Bulk Vector Register Access Timing for a Write Followed by a Read of a 7-Element Vector.	107
Figure 5.7: Generalized Fast/Bulk Vector Register Supporting Simultaneous Read and Write Access.	112
Figure 5.8: Internal Details of a Bulk-RAM Bank.	114
Figure 5.9: Sequence of Delays for Read Access.	116
Figure 5.10: Sequence of Delays for Write Access.	117
Figure 5.11: Placement of Pipelined-Propagation Latches for an Eight-Region Pipe- lined RAM Chip.	120
Figure 5.12: Fast/Bulk Vector Register with 64 Elements, Implemented Using Pipe- lined RAM Chips.	123
Figure 6.1: Code Sequence Producing a Start-Limit of Eight.	127
Figure 6.2: Operation Timing for Code Sequence Producing a Start-Limit of Eight.	128
Figure 7.1: Instruction Formats of the Cray-2.	151
Figure 7.2: Instruction Formats of the Larc.	156

List of Tables

Table 3.1: Livermore Kernels 1-14, with Iteration Counts and Operation Counts.	30
Table 3.2: Eager, Lazy, and Scheduled Partitioning and Critical Path for the Exam- ple in Figure 3.1.	36
Table 3.3: Partition Sizes and Critical-Path Widths.	42
Table 3.4: Fraction of Memory References with Data-Independent Addresses.	43
Table 3.5: Result Lifetimes (in Partitions) for Short-Result-Life Scheduling and Eager-Execution Scheduling.	44
Table 3.6: Number of Times Each Result Used.	45
Table 3.7: Root-Mean-Square Differences Between Execution Simulations and Bounds Set by Theorems 3.6 and 3.7, for Livermore Kernels 1-14.	48
Table 3.8: Number of Fast Registers Required and Number of Times Fast Registers Used (Reservation from Write to First Result Read, Slow-Register Read and Write Penalties = 1).	59
Table 3.9: Number of Fast Registers Required and Number of Times Fast Registers Used (Reservation from Start to First Result Read, Slow-Register Read and Write Penalties = 1).	61
Table 3.10: Times for Simulated Execution with Unlimited Fast Registers and with Few Fast Registers and Unlimited Slow Registers (Slow-Register Read and Write Penalties = 1).	62
Table 3.11: Number of Fast Registers Required and Number of Times Fast Regis- ters Used (Reservation from Start to First Result Read, Slow Register Read	

Penalty = 2, Slow Register Write Penalty = 1).	63
Table 3.12: Times for Simulated Execution with Unlimited Fast Registers and with Few Fast Registers and Unlimited Slow Registers, Slow Register Read Penalty = 2, Slow Register Write Penalty = 1.	63
Table 3.13: Number of Fast Registers Required and Number of Times Fast Regis- ters Used (Reservation from Start to First Result Read, Slow Register Read Penalty = 4, Slow Register Write Penalty = 1).	64
Table 3.14: Times for Simulated Execution with Unlimited Fast Registers and with Few Fast Registers and Unlimited Slow Registers, Slow Register Read Penalty = 4, Slow Register Write Penalty = 1.	64
Table 4.1: Propagation Speeds and Delays for Various Media.	68
Table 4.2: Mapping of Address Bits to Devices.	76
Table 4.3: Delays for Fanning Out, Multiplexing, and Decoding.	78
Table 4.4: Smallest Numbers of Levels of Logic n that Require No Extra Delays, for Various Gate-Delay Uncertainties r	81
Table 4.5: Optimal Numbers of Levels of Logic p per Stage and Numbers of Stages S for Various Numbers of Independent Operations N , with Corresponding Clock Periods C and Latencies L when the Logical Function Requires 50 Lev- els of Logic and the Gate Speed Uncertainty is 0.3.	84
Table 4.6: Processing Times for Various Quantities of Data N and for Various Numbers of Stages S	88
Table 4.7: Processing Ratios for Various Quantities of Data N and for Various Numbers of Stages S	88
Table 4.8: Weighted Mean Processing Ratios for Various Number of Stages, Assum- ing Uniformly Distributed Workload.	89

Table 4.9: Numbers of Instances of Groups of Parallel Operations of Each Range of Sizes, with Weights.	89
Table 4.10: Weighted Mean Processing Ratios for Various Numbers of Stages S , As- suming the Workload Distribution of Livermore Kernels 1-14.	90
Table 5.1: Numbers of Levels of Logic Required if All Inputs are Multiplexed Down to One Output, for Fan-in = 5, Up to 16 Other Functional Unit Inputs, with Various Numbers of Close Registers, and for Various Values of Bandwidth.	101
Table 5.2: Instruction Stream Bandwidth Requirements in Bits per Operation for Close/Distant Registers and for a Set of 8 Registers, for Livermore Kernels 1-14.	104
Table 5.3: Fraction of Area Overhead Due to Pipeline Latches, as a Function of the Number of Bits.	118
Table 6.1: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the Cray-1.	129
Table 6.2: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Cray-1.	131
Table 6.3: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the Cray X-MP.	133
Table 6.4: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Cray X-MP.	134
Table 6.5: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Cray-2.	138
Table 6.6: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the NEC SX-2.	140

Table 6.7: Simulated Execution Rates for Livermore Kernels 1-14	
Executing on the NEC SX-2.	142
Table 6.8: Simulated Execution Rates for Livermore Kernels 1-14	
Executing on the HEP-1.	145
Table 6.9: Operation Types and Actual Start-Limits for Vectorizable Livermore	
Kernels Executing on the ETA-10.	148
Table 6.10: Simulated Execution Rates for Livermore Kernels 1-14 Executing on	
the ETA-10.	148
Table 7.1: Simulated Execution Rates for Livermore Kernels 1-14	
Executing on the Larc.	160
Table 7.2: Cray-2 Instruction Set.	161
Table 7.3: Larc Instruction Set.	164
Table C.1: Execution Time Simulations and Bounds for Livermore Kernel 1, with	
Differences Between Simulated and Theoretical Times.	191
Table C.2: Execution Time Simulations and Bounds for Livermore Kernel 2, with	
Differences Between Simulated and Theoretical Times.	192
Table C.3: Execution Time Simulations and Bounds for Livermore Kernel 3, with	
Differences Between Simulated and Theoretical Times.	193
Table C.4: Execution Time Simulations and Bounds for Livermore Kernel 4, with	
Differences Between Simulated and Theoretical Times.	194
Table C.5: Execution Time Simulations and Bounds for Livermore Kernel 5, with	
Differences Between Simulated and Theoretical Times.	195
Table C.6: Execution Time Simulations and Bounds for Livermore Kernel 6, with	
Differences Between Simulated and Theoretical Times.	196

Table C.7: Execution Time Simulations and Bounds for Livermore Kernel 7, with Differences Between Simulated and Theoretical Times.	197
Table C.8: Execution Time Simulations and Bounds for Livermore Kernel 8, with Differences Between Simulated and Theoretical Times.	198
Table C.9: Execution Time Simulations and Bounds for Livermore Kernel 9, with Differences Between Simulated and Theoretical Times.	199
Table C.10: Execution Time Simulations and Bounds for Livermore Kernel 10, with Differences Between Simulated and Theoretical Times.	200
Table C.11 Execution Time Simulations and Bounds for Livermore Kernel 11, with Differences Between Simulated and Theoretical Times.	201
Table C.12: Execution Time Simulations and Bounds for Livermore Kernel 12, with Differences Between Simulated and Theoretical Times.	202
Table C.13: Execution Time Simulations and Bounds for Livermore Kernel 13, with Differences Between Simulated and Theoretical Times.	203
Table C.14: Execution Time Simulations and Bounds for Livermore Kernel 14, with Differences Between Simulated and Theoretical Times.	204

Index of Definitions

<i>Start-limit</i>	37
<i>Etime</i>	37
<i>Opcount</i>	37
<i>First</i>	37
<i>Earliest</i>	37
<i>Last</i>	37
<i>Latest</i>	37
<i>Minstarts</i>	40
<i>Mazstarts</i>	40
<i>Overlap</i>	41
<i>Starts</i>	41
<i>Time</i>	41



CHAPTER 1

Introduction

1.1. Problem Description

Most computations produce many temporary results in the course of execution. For example, if the statement $W = X * Y + Z$ is executed on a load-store architecture, it might be compiled into a code sequence similar to that shown in figure 1.1.

```
R1 ← mem[X]
R2 ← mem[Y]
R3 ← R1 * R2
R4 ← mem[Z]
R5 ← R3 + R4
mem[W] ← R5
```

Figure 1.1: Hypothetical Code Sequence for $W = X * Y + Z$.

The temporary results of the loads or the multiply may be of no interest to the programmer, but they are necessary for correct execution of the program, and the efficiency of their storage can have a significant impact on the running time of the program.

The critical path of the code in figure 1.1 includes a load, a multiply, an add, and a store. The load time includes the time to fetch either X or Y from memory and write it into $R1$ or $R2$. The multiply time includes the time to read $R1$ and $R2$, compute their product, and write the result to $R3$. The add time includes the time to read $R3$ and $R4$, compute their sum, and write it to $R5$. The store time includes the time to read $R5$ and copy it to W in memory. The time to execute the code sequence is at least the sum of times to execute each of the operations in the critical path; this includes three register read times and three register write times.

In this code sequence, at most three operations can execute concurrently, and most of the time only one or two operations can execute concurrently. Thus, this code sequence has little parallelism: it is relatively serial instead.

The inner product of two 1024-element vectors $\sum_{i=1}^{1024} (X[i] * Y[i])$ can be computed using a binary summation tree to add all the products. The 2048 load operations can execute concurrently, the 1024 multiply operations can execute concurrently, and, for each level of the summation tree, the add operations can execute concurrently. A program that uses a binary summation tree to compute the inner product is very parallel.

Parallel execution of this inner product can require thousands of temporaries. If the temporaries are stored in memory, the time to perform the computation includes the time to store and then load the temporary results at each level of the summation tree. In addition, the program must include explicit store and load instructions that store and then load the temporary results.

If the temporary results are stored in registers, rather than in memory, the extra load and store instructions can be avoided; in addition, temporaries can be accessed much faster from registers than from memory. However, accessing one of thousands of registers is slower than accessing one of a few registers, because data must propagate over physically longer paths between registers and functional units, and they must be routed through more levels of logic when they are selected. Furthermore, the addresses for many registers are longer than for few registers, requiring more instruction-stream bandwidth.

Thus, it appears that if an architecture is capable of executing many operations concurrently, the optimal choice of the number of registers depends on the characteristics of the programs running on an architecture: serial applications need few registers (and would in fact run slower with many registers than with few registers), while parallel applications need many

registers. The apparent conflict between the temporary storage requirements for serial applications and the requirements for parallel applications is the subject of this dissertation.

To resolve the conflict, this dissertation describes an approach in which registers are implemented and used asymmetrically. A small set of registers that can be accessed with low latency is used for serial programs or serial sections of programs, while a larger set of registers that have longer-latency, high-bandwidth access are used for parallel programs or parallel sections of programs. Instruction-stream bandwidth requirements for addressing large sets of registers are reduced by structuring their addressing mechanisms in ways that match the temporary result usage of scientific programs, addressing many of the registers implicitly. Temporary results are allocated to the asymmetrical registers using a straightforward procedure based on critical-path analysis of programs.

1.2. Thesis Statement

The traditional use of memory and a symmetrical set of registers for storage of temporary results of scientific programs requires more execution time, hardware, and instruction-stream bandwidth than necessary. Novel register organizations that can be easily integrated into traditional supercomputer architectures can reduce all of these requirements.

Execution speed can be more than doubled by storing temporary results in an asymmetrical set of general-purpose registers or an asymmetrical set of vector registers, instead of in memory and a small register set. Faster access and a hardware cost one fourth that of traditional vector registers can be had by using a vector register that incorporates a pipelined, random-access-memory chip. If a large enough set of registers is used, the need to store temporary results in memory and then reload them for later use can be eliminated; this saves both instruction-stream bandwidth and execution time.

1.3. Approach

The claims in the previous section are proved in four steps. A set of benchmark programs is analyzed to determine the hardware requirements for fast execution. Datapath restrictions are analyzed to show why these hardware requirements cannot be met using traditional symmetrical register structures and to show how asymmetrical organizations can be used to circumvent some of the restrictions. Asymmetrical storage structures that take advantage of these analyses are synthesized to show how practical storage structures can be implemented. Finally, an existing supercomputer architecture is modified, to incorporate new datapath structures and storage structures, to show that the modification can be easily done and to show the impact on performance.

1.4. Research Contributions

The research reported in this dissertation provides the following contributions:

- (1) A procedure, based upon a hardware-independent partitioning of the program dependency graphs, is presented for analyzing programs. The hardware requirements and execution times of programs can be estimated from the partitionings of their dependency graphs, and analyses of these condensed specifications of programs are both faster and more generally applicable than analyses of the full dependency graphs. This analysis procedure is used to determine the proper architectural parameters for scientific program execution.
- (2) Analyses and simulations of the Livermore Kernels⁶⁷ show that execution with a small set of fast registers and a large set of slower registers is as fast as execution with a large set of fast registers. At least 256 registers are necessary for many programs to run as fast as they can with an unlimited number of registers; this is often more than a factor of two faster than execution using only eight registers.

- (3) An analysis of pipelined execution shows that the optimal number of levels of logic in each pipeline stage grows as the square root of the number of levels of logic to perform the function, divided by the square root of the number of independent data to be processed at a time. If several identical pipelines are used together, the optimal number of levels of logic in each stage decreases by the square root of the number of identical pipelines. This analysis allows pipelined functional units to be designed so that their latency/throughput tradeoffs match the characteristics of particular workloads.
- (4) An analysis of pipelines with variable bandwidth and latency shows that they are not practical to implement. The logic necessary to control the number of clocked pipeline stages increases the pipeline latency more than switching out stages reduces it.
- (5) An organization of registers into a close/distant hierarchy allows some registers to be accessed quickly, while other registers are accessed more slowly. The access bandwidth of the close/distant registers is as high as that of a set of registers that are all fast.
- (6) Vector registers are presented that provide high-bandwidth access and high-density storage of all elements, with fast access to some elements. Relative to traditional vector registers, these new vector registers are longer for the same latency and bandwidth; alternatively, they provide lower-latency, higher-bandwidth access for the same length. In addition, they require one fourth the hardware for the same length and bandwidth.
- (7) A novel pipelined, random-access-memory organization is described. It increases the bandwidth of access by a factor of four over traditional random access memory, costing less than a 50% increase in access latency and less than a 50% reduction in storage density. This pipelined, random access memory is more advantageous than interleaved memory because its performance is independent of access pattern. It is particularly well-suited to vector register implementations and for memory systems that are traditionally implemented using interleaved memory banks.

- (8) An improved supercomputer architecture using an asymmetrical organization of registers and using the new vector registers is specified.

1.5. Terminology Conventions

An *operation* is an atomic transformation of the architectural state. It takes one or more operands and produces a *result* visible to the programmer. Examples of operations include loads from memory to general-purpose registers, logical ANDs, floating-point multiplies, and conditional branches.

A *micro-operation* is an atomic transformation of the microarchitectural state, which may or may not be visible to the programmer. An example of a micro-operation that is programmer-invisible is the fraction add of a floating-point add operation, which changes the state of one or more pipeline stages but does not produce a result that can be directly manipulated by other operations. An example of a micro-operation that does change the state of the architecture is copying the output of the floating-point adder to a general-purpose register.

An *instruction* is the specification of one or more operations to be performed. Examples of instructions are a simple add instruction, which specifies one add operation to be performed, and a vector load instruction, which usually specifies many load operations to be performed.

In the context of circuits, a *latch* is a logical circuit with feedback; when the data inputs are enabled (usually by one or more clock inputs), the output is a combinational function of the inputs, and when the data inputs are disabled, the output is a function independent of the current inputs. Latches are distinguished from master-slave flip-flops, which can be constructed using two latches and for which the output does not change except at the transition from enabled data inputs to disabled data inputs.

In the context of a microarchitecture, a latch is a storage device that is programmer-invisible. Latches are situated between stages of all pipelines.

A *register* is a programmer-visible storage element that has fast access and a relatively short address. The program counter and general-purpose registers of a traditional architecture are examples of registers. All the elements of a vector register together constitute the register. Registers can be implemented using either latch circuits or master-slave flip-flops.

A *device* is any circuit element that can contain data of interest. Examples of devices are latches, functional units, and registers.

A *load* operation copies the contents of a memory location to a register, and a *store* operation copies the contents of a register to a memory location.

A *write* is a micro-operation where a datum is copied to a register. A *read* is a micro-operation where the contents of a register are copied to some other device.

A *clock tick* is the minimal time interval for signal synchronization; it is limited by the maximum time required for data to propagate from the output of a latch, through wires and zero or more levels of logic, to the input of another latch, and to become stable at that latch's output. Typically, a clock tick (or simply tick) is long enough to allow from four to 25 levels of logic between latches. Ticks are usually measured in units of the basic gate-delay for the logic technology used.

1.6. Organization of this Dissertation

In Chapter 2, previous work that is relevant to supercomputer design is surveyed. Work that this dissertation builds upon is touched upon here, but is analyzed in detail in the chapter in which it is used.

In Chapter 3, the program characteristics of supercomputer applications are analyzed. The notion of dependency graph partitioning is introduced as a means to analyze the pro-

grams. Bounds are established on the execution times of programs and on the number of storage locations that are required. The effects of various temporary storage characteristics on program execution time are measured by simulation. The results of these analyses and simulations are used to determine the architectural requirements of supercomputers for scientific program execution.

In Chapter 4, datapath designs are analyzed, including limitations on clock speed, device selection time, latch design, and pipeline latency/throughput tradeoffs. Earlier results in datapath design are analyzed in detail. An asymmetrical allocation of devices to circumvent some of these limitations is presented.

In Chapter 5, novel storage devices and access mechanisms are introduced. An asymmetrical-register-access mechanism is presented, along with an asymmetrical-register-allocation procedure. Several designs for asymmetrical vector registers are presented, which provide a range of capabilities, including those of the Cray-1,²¹ the Cray X-MP,²² the NEC SX-2,⁴⁰ as well as capabilities in excess of all of these. A design for a high-bandwidth random access memory (RAM) with pipelined access is presented. This pipelined RAM provides two to four times the bandwidth of traditional RAMs with similar technological parameters; this costs one additional input pin, up to a 50% increase in access latency, and up to a 50% reduction in storage density. This pipelined RAM is well-suited for asymmetrical vector register implementations and those applications that traditionally use interleaved memory.

In Chapter 6, several existing supercomputer designs are critically analyzed, including the Cray-1,^{21,54} the Cray X-MP,²² the Cray-2,²³ the ETA-10,^{33,34,84} the HEP-1,²⁶ and the NEC SX-2.^{40,49,106} The emphasis is on the functional unit organization and on the characteristics of storage for temporary results.

In Chapter 7, the architectural integration of the storage structures for temporary results is demonstrated by an example of a supercomputer architecture that incorporates the

asymmetrical-register-access mechanisms and vector register design discussed in Chapter 5.

This architecture is a modification of the Cray-2 architecture, analyzed in Chapter 6.

In Chapter 8, results are summarized, conclusions are drawn, and future work is discussed.

CHAPTER 2

Survey of Previous Work

While this thesis focuses on the temporary storage aspects of supercomputer design, many other areas must also be considered. This chapter is devoted to discussions of the work applicable to optimization and scheduling, application characteristics and program structure, datapath design issues, storage design issues, instruction issuing and instruction formats, and some of the past and current supercomputers.

The results that this dissertation directly use are touched upon in this chapter, but are analyzed in detail in the chapters that use them. With this organization, the necessary terminology and definitions can be introduced before the results are analyzed; this improves both brevity and clarity.

2.1. Optimization and Scheduling

Optimization and scheduling can have a greater influence on the execution speeds of programs for supercomputers than for slower computers. Programs for supercomputers with vector instructions must be run through a vectorizer to determine which operations should be performed in vector mode and which should be performed in scalar mode. This is in addition to the general optimization techniques applied to all computers. Supercomputers also require that operations be scheduled for these reasons: to avoid dependency conflicts due to overlapped execution, and to avoid hardware resource conflicts among the multiple functional units and vector functional units.

2.1.1. Optimization

Most program transformations do not produce optimal code, because the process is NP-complete in most cases. Although code is only improved, the term optimization has been used universally.

Muchnick and Jones⁷¹ described general program optimization techniques in their book on program flow analysis, and Chow¹⁶ described a portable, global optimizer in his PhD thesis. Both of these works concentrate on the issues of optimization of programs executing on serial computers; while many of the techniques described can be applied to supercomputers, some (such as evaluation-stack-height reduction) tend to slow execution on many supercomputers (assuming a sufficient quantity of registers, what is wanted is *tree-height* reduction; stack-height reduction tends to increase the height of the expression tree for parallel or overlapped execution). David Kuck's excellent text on computer architecture⁵⁵ describes optimization techniques that are applicable to supercomputers and parallel architectures. These include tree-height reduction, solving linear recurrences to increase execution parallelism, and data and control dependence transformations to allow more parallel execution.

Agerwala² surveyed the state of microcode optimization in 1975, and noted that much of the effort concentrated on optimal solutions using exhaustive methods, rather than on practical engineering solutions. Landskov, et.al.⁶¹ discussed local microcode-compaction algorithms, including linear compaction, critical-path compaction, branch-and-bound techniques, and list scheduling. Garey and Johnson⁴² discussed the NP-completeness aspects of program and microprogram optimization. Most optimizations are NP-complete, although for a few specific cases (such as determining register sufficiency for loops), polynomial-time solutions exist. The NP-completeness of optimization is significant because the problem sizes are large; hundreds or thousands of operations must be scheduled for optimization to be effective.

2.1.2. Scheduling

Scheduling resources is an extremely difficult problem. French's text³⁰ provides a very readable introduction to job-shop scheduling. Gonzalez⁴⁴ surveyed deterministic processor scheduling of uniprocessors and multiprocessors, concluding that efficient, optimal algorithms exist for only a few special cases and suggesting that further efforts be directed towards the study of heuristics. He advocated the use of statistical methods as a new approach to the problem. Shapiro⁸⁷ considered the specific task of scheduling tasks that are coupled in time, with the constraint that the scheduling must be performed in real time. The sub-optimal methods used included sequencing, nesting, and fitting. He found that sequencing (the fastest method to schedule) produced results almost as good as fitting (the slowest method to schedule) and considerably better than nesting.

2.1.2.1. Static Scheduling

Program scheduling is NP-complete, and, although small loops have been optimally scheduled, most effort has been directed towards developing good heuristics.

Arya^{5,6} showed that scheduling pipelined processors (in particular, the Cray-1 S) is NP-complete, and he presented an optimal solution using integer linear programming. He transformed the scheduling problem into an integer-linear-programming problem, and then used a commercial, integer-linear-programming package to find a solution. His proof that the scheduling problem cannot be solved using normal linear programming was similar to that described by Papadimitriou and Steiglitz⁷⁵ in their book on combinatorial optimization; Arya's transformation of the problem to an integer-linear-programming problem was also similar to that described in their book.

Much of the work in pipeline-processor scheduling has been in the development of efficient heuristics. Berlekamp described his personal techniques for loop optimization on the CDC 6600,⁸ and described a program used to support loop tuning.⁹ Nelson wrote a time-

tabling program⁷² to aid in speeding up programs on the Cray-1. Weiss and Smith¹¹⁰ described scalar compilation techniques for the Cray-1 scalar unit. Their techniques included loop unrolling and software pipelining, and their approaches were similar to those described by Berlekamp for the CDC 6600 and to those described by Foster and Riseman for the CDC 3600. Weiss and Smith claimed execution rates comparable to that of the vector unit of the Cray-1 with the Cray vectorizing compiler for Livermore Kernels 1-14.⁶⁷ However, their version of the compiler was obsolete at the time, because it failed to vectorize kernels that CIVIC version 131e²⁴ was able to vectorize. Furthermore, they reported speedups of at most 3.2 over execution without the use of vector instructions, and they did not compare execution speed to execution using vector instructions.

Foster and Riseman³⁸ described their experiences in code percolation as a means to avoid issue stalls, noting that it achieved 93.5% of the parallelism achieved by an infinite issue stack, at least for their programs for the CDC 3600. They also reported that conditional branches tend to inhibit potential parallelism,⁸⁵ without the conditional branches, they found concurrency on the order of 50 instructions.

Kohler⁵³ described a general heuristic for scheduling that concentrated on the critical paths of programs. He reported results within a factor of two of optimal.

Rau, et.al.⁸³ described a novel use of memories within the crosspoint switches between functional units. These memories hold results until they are used, and they drastically reduce the complexity of static operation scheduling for a microprogrammed processor. With the new memories, operations can be scheduled to execute any time after their operands are available, rather than exactly when their operands become available.

2.1.2.2. Dynamic Scheduling

Dynamic scheduling necessarily must limit the window in which instructions are scheduled, but it can use dynamic information to improve the quality of schedules. Much of

the effort has concentrated on minimizing the hardware required to schedule operations.

Keller⁵¹ surveyed instruction issuing techniques as they apply to dynamic scheduling, and established bounds on sizes and complexities of the scheduling hardware.

Thornton^{99,100} described the scoreboard mechanism of the CDC 6600 that allows instruction execution to overlap without violating instruction-dependency constraints. Tomasulo¹⁰³ described his famous algorithm for the IBM System/360 Model 91 floating-point unit that allows instruction result registers to be dynamically reassigned during execution. Weiss and Smith¹⁰⁹ investigated several dynamic scheduling strategies, ranging from that used in the Cray-1 to the Tomasulo algorithm. The Tomasulo algorithm is the most complex and produces the fastest execution, while the strategy used for the Cray-1 is the least complex and produces the slowest execution. The complexity and performance used in Weiss and Smith's own strategy was between these extremes. Acosta, et.al.¹ proposed dynamic scheduling hardware that allows parallel issuing of instructions. For Livermore Kernels 1-14, they reported speedups of 1.7-2.8 over serial-dispatch schemes.

Nonlinear pipelines, which use some pipeline stages more than once for each operation, must be scheduled so that different operations do not request the same resources at the same times. An excellent introduction to pipeline scheduling theory is provided in Peter Kogge's book on pipelined architectures.⁵² Shar's PhD thesis⁸⁸ discusses static pipelines (those that perform only a single set of functions at a time), and Thomas' PhD thesis⁹⁸ discusses dynamic pipelines (those that can be reconfigured while performing operations).

2.2. Application Characteristics and Program Structure

Supercomputer applications tend to have many floating-point operations and some degree of parallelism. They differ in distributions of operations, degrees of parallelism, and memory usages.

Probably the best known set of supercomputer benchmarks is the collection of Livermore Kernels 1-14,⁶⁷ program kernels characteristic of the workload at Lawrence Livermore National Laboratories. These are discussed in more detail in Chapter 3. Other benchmarks include: Dongarra's LINPACK,²⁹ a collection of linear algebraic codes for matrix decomposition and matrix-vector multiplies; BMK,⁵⁸ a collection of benchmarks characteristic of the workload at Los Alamos National Laboratory; Simple,⁵⁹ an unclassified hydrodynamics code; and EISPACK,⁴¹ a collection of codes for computing eigenvalues and eigenvectors. Two other benchmarks, Whetstone²⁵ and Dhrystone,¹⁰⁸ are of little interest because they are totally synthetic and bear no similarity to actual programs, other than the authors' assumed frequencies of various types of operations. Nothing in the way of operation distributions or address sequences can be determined or predicted from Whetstone or Dhrystone.

Other typical supercomputer applications include Fourier transforms, matrix transformations, image generation, and mathematical function evaluation. An excellent introduction to the fast Fourier transform is found in the text by Brigham.¹³ It includes discussions of Fourier transforms, discrete Fourier transforms, and fast Fourier transform algorithms for arbitrary factors. Lambiotte and Voight⁶⁰ discussed transformations of the obvious serial algorithm for tridiagonal-matrix factorization into more parallel algorithms that perform extra work, but which can execute faster on pipelined computers with long-latency operations. If sparse matrices are factored using Gaussian elimination, they become full and can no longer fit in main memory. Nour-Omid and Parlett⁷³ discussed conjugate-gradient methods for sparse-matrix manipulations that do not generate full matrices but that require more arithmetic operations than does Gaussian elimination. With proper element preconditioning, conjugate-gradient methods converge on solutions faster than Gaussian elimination methods that produce matrices that do not fit in main memory. Dippé and Swensen²⁷ discussed the problem of image generation, with an emphasis on ray-tracing methods. They described a parallel algorithm for ray-tracing that reduces the total number of operations performed.

Elementary mathematical function implementation is discussed in Cody and Waite's text on the subject,¹⁷ with an emphasis on trigonometric, log and antilog functions for fixed-point, floating-point, decimal, and non-decimal machines. Many more numerical algorithms are discussed in the text by Press, et.al.,⁷⁹ but with fewer details than Cody and Waite's text provides. Kung⁵⁶ studied several types of scientific programs, noting that the memory size must grow much faster than the processing speed increases if input and output time is to remain constant.

The structure of program parallelism has been discussed by several authors. Berlekamp discussed critical and noncritical paths in loops executing on the CDC 6600, and showed how to use this information to efficiently schedule instructions.^{8,9} Tjaden and Flynn¹⁰¹ analyzed programs for the IBM 7094, and described three kinds of dependencies: data dependencies that are caused by one instruction using another's result, procedural dependencies that are caused by an instruction following a branch that depends on another instruction's result, and operational dependencies that are caused by resource-usage conflicts. They found that an 86% improvement in performance was possible by issuing instructions in parallel. Foster and Riesenman^{38,85} showed how several independent chains of dependent instructions can be interleaved (or percolated), allowing many instructions to issue without stalling and to execute in parallel. They found that data-dependent conditional branches were the major obstacle to parallel execution; if an oracle could predict their outcome, as many as 51 instructions could execute in parallel, on the average. Tjaden and Flynn¹⁰² described a system of ordering matrices to represent dynamic program concurrency. Their simulations show moderate speedups when these matrices are used. Lambiotte and Voight⁶⁰ analyzed the performance of several algorithms for factorization of tridiagonal matrices on pipelined computers. They found that by performing extra work (beyond that required by Gaussian elimination), the dependency chain present in the obvious Gaussian-elimination algorithm can be broken; in addition, they presented efficient algorithms for asymptotic cases, as well as for smaller matrices.

Bucher¹⁴ analyzed the performance of several vector operations on the Cray-1 S and on the Cyber 205. She noted that the slowest characteristic execution speed of a computer will become the performance bottleneck, unless that characteristic represents a negligibly small fraction of the total workload.

Berlekamp described parallel operations in Galois field computations in the patent disclosure for his Galois field computer.¹⁰ Dippé and Swensen²⁷ described several forms of parallelism present in computing ray-traced computer graphics images. They showed that space subdivision of the problem requires far less computation than time subdivision.

Acosta, et. al.¹ discussed a hardware scheme to maintain dynamic dependency information about programs. When there is sufficient parallelism, several instructions can be issued in parallel.

Thornton¹⁰⁰ described features included in the CDC 6600 to support multiprogramming, specifically base and limit registers, a fast exchange jump, and several peripheral processors. Larson⁶² described multi-tasking support hardware in the Cray X-MP, including shared memory and inter-processor communication registers. A different approach is taken by the HEP-1,²⁶ which supports the concurrent execution of several processes at once, and has a synchronization mechanism built into the memory system.

2.3. Datapath-Design Issues

2.3.1. Pipelining

Most of the recent supercomputers have been pipelined, and work has been done on pipeline clocking and control, latch design, and hardware allocation.

Peter Kogge's definitive text on pipelining⁵² discusses all aspects of pipelined computer design, from components and clock circuits to pipelined architectures and instruction sets. Blaauw also discussed pipelining to a limited extent in his text on functional unit

implementation.¹¹ Ramamoorthy and Li presented a pipelining survey, with a particularly good comparison of the Control Data Corporation Star-100 and the Texas Instruments Advanced Scientific Computer.⁸⁰

Shar⁸⁸ and Thomas⁹⁸ discussed optimal control strategies for nonlinear, pipelined functional units. Cotten discussed clocking and control of pipelines,¹⁹ and also proposed the more general "maximum-rate" pipelining,²⁰ where a pipeline can contain more data than it has latches.

Hallin and Flynn discussed pipelining of arithmetic functions,⁴⁶ and advocated use of a latch designed by Earle to achieve high computational-efficiency. Kunkel and Smith⁵⁷ discussed the clocking requirements of various pipelined circuits, and investigated the optimal number of logic levels between pipeline stages. For the Livermore Kernels and functional units like those in the Cray-1, they concluded that between eight and ten levels of logic between stages is optimal.

2.3.2. Arithmetic Unit Design

The work in arithmetic unit design has concentrated on minimizing the execution times for expected workloads and efficient partitioning of hardware.

Blaauw described most of the important hardware algorithms for arithmetic units in his text on digital systems,¹¹ including algorithms for addition and multiplication that speed up carry propagation. He used APL as an implementation-independent language for presentation. Swartzlander has collected many of the landmark papers in computer arithmetic into one volume.⁹⁴

Cotten^{19,20} discussed the clocking, control, and logic partitioning of pipelined functional units. Shar⁸⁸ and Thomas⁹⁸ discussed the control of pipelined arithmetic units, and showed how to achieve maximum utilization of the hardware. Kunkel and Smith⁵⁷ compared several

partitionings of arithmetic unit functionality for the Cray-1 running the Livermore Loops, and concluded that between eight and ten gates of logic between logic stages is optimal for that architecture and problem set.

Thornton described the specific arithmetic units in the CDC 6600 in his wonderful book about that computer.¹⁰⁰ Anderson described some aspects of the floating-point execution unit of the IBM System/360 Model 91 computer,⁴ and Watson¹⁰⁷ described the Texas Instruments Advanced Scientific Computer, with its four homogeneous arithmetic pipelines that can be configured to perform the functions needed by the applications. Lincoln⁶⁴ presented an interesting comparison between the CDC Star-100 and its successor, the Cyber 205, citing technology as a major reason for changing the design. Finally, Russell⁸⁶ described some aspects of the Cray-1 arithmetic units, noting that the short pipelines allow vector execution to be efficient even for short vectors.

2.3.3. Decoding Delays

In his text on computer architecture,⁵⁵ Kuck showed that $\log_f(k)$ levels of devices are needed to decode or multiplex one of k things, where f is the fan-in or fan-out of each device. Meade⁶⁸ noted that larger memories tend to have longer access times than smaller memories, because the former have longer cables and, hence, longer propagation delays. Tjaden and Flynn¹⁰¹ noted that logic decoding time increases with n , where n is the number of instructions simultaneously issued each clock tick. Thus, the length of each clock tick tends to increase as more instructions are simultaneously issued.

2.3.4. Components

Supercomputer design has always been driven by advances in components.^{31,32,85} Thornton^{99,100} discussed the circuits and three-dimensional "cordwood" logic modules used in the CDC 6600. Lincoln⁶⁴ attributed the refinement of the transistor and the cordwood module

with the success of the CDC 6600. He noted the use of freon cooling to allow more dense packaging of the logic. He also discussed the limitations that small scale integrated (SSI) circuits imposed upon the CDC Star-100's design, where functionality had to be limited in order to keep the parts count small enough (for reliability reasons), and he discussed how the availability of large scale integrated (LSI) circuits enabled the Cyber 205 to take its final form. He also described numerous problems that LSI circuits caused, particularly those problems caused by too few logic module types.

Russell⁸⁶ described the four chip types used in the Cray-1 (two speeds of OR/NOR gates, a 16×4 RAM chip, and a $1K \times 1$ RAM chip), and discussed at length the power distribution and cooling system used for this computer; in fact, he stated that the cooling system is the only part of the Cray-1 that is patented.

Bloch¹² discussed component advances and their influence on computers, as did Vacca,¹⁰⁵ who concentrated on packaging issues of components and logic partitioning.

2.4. Storage-Design Issues

Much work has been done in memory system design, including caches, interleaving, virtual memory, and the tradeoffs between caches and registers.

Pohm and Agrawal⁷⁷ discussed many aspects of memory design in their text, including a history of cache memories, swapping algorithms, effective cycle-times and access-times, hit ratios, and organizations of memory hierarchies. Smith⁹¹ surveyed cache memories, with an emphasis on caches for the large IBM and Amdahl computers. He noted that cache bandwidth should be at least two or three times the average data-rate in order to be effective.

Takahashi, et.al.⁹⁷ described the memory system of the ETL Mk-6 developed in the Electrotechnical Laboratory in Tokyo; the top section of the arithmetic stack in the ETL Mk-6 is implemented in fast tunnel-diode memory. Meade⁶⁸ advocated mapping the "relative

value" of data to various levels of the memory hierarchy, so that frequently-accessed data are available faster than less-frequently-accessed data. Both of these ideas reflect the conventional wisdom that frequently-accessed data should be available quicker than less-frequently-accessed data. What they ignore, however, is the tradeoff between the latency and bandwidth of access.

Lincoln⁶⁴ advocated virtual memory for supercomputers, citing as reasons: the easier task of memory management, convenient debugging of large programs, and easier development of large programs. The computers of Seymour Cray, however, do not have virtual memory, but are much more common in supercomputer installations. This seems to suggest that convenience of use is not an overriding consideration for the institutions that purchase supercomputers.

Smith, et.al.^{92,93} described a new architecture that decouples memory access mechanisms from execution mechanisms. Their architecture compares favorably to the Cray-1 on scalar compilations of the Livermore Loops, but it is not clear from their studies how their architecture compares to vector execution of the same loops.

Kung⁵⁶ discussed the memory requirements for multiprocessors with private memory. He considered several types of scientific computations when inter-processor communication rates are held constant and execution rates are increased by a factor of α . For matrix computations, memory size must be increased by a factor of α^d (where d is the dimensionality of the matrix or mesh). For FFT or sorting computations, the memory size required is that of the original memory raised to the power of α . If memory size is not increased, inter-processor communication time will dominate the total time for the computation.

Many studies of interleaved-memory-bank contention have been performed. Rau⁸² noted that the memory bank interference problem is the dual of paging; page "hits" are desired in a virtual memory system, while "faults" are wanted to avoid memory bank

interference. He developed an access model based on a least-recently-used (LRU) replacement policy that is superior to random models; he advocated a cache to "sequentialize" accesses to memory banks (and hence decrease their hit rate), and noted that, for random memory bank requests, deep interleaving performs no better than shallow interleaving.

Oed and Lange⁷⁴ reported minimum-conflict access patterns for the Cray X-MP memory system using GCD methods to prove their results. Unfortunately, they did not show how to improve the performance of the memory system when access patterns are fixed by the program. They showed several access patterns with linked conflicts, where one access pattern forces another to wait, causing the patterns to interfere with each other in the same way again. Cheung and Smith¹⁵ investigated bank conflicts for the Cray X-MP, and found all bank conflicts for all access patterns with unit stride. They showed that having a busy time for a bank that is different than the number of sections results in fewer linked conflicts. (A sections is a collection of memory banks that are independent of each other, but that share common data paths to and from the CPU.) They advocated mapping the memory sections selection to bits 3 and 2 of the address, rather than bits 1 and 0, citing a 7% reduction in sections conflicts for access patterns with unit stride when this is done. (A sections conflict occurs if two or more requests to different banks in the same sections arrive simultaneously.)

Bailey⁷ simulated memory bank conflicts for vector access patterns, and reported that deep interleaving substantially reduces memory bank interference for sequential vector access patterns. He also reported a large increase in bank interference when vector strides greater than unity are used. Note that Bailey's results, which deal with sequential access, are not in conflict with Rau's results, which deal with random access.

Swensen and Patt⁶⁶ proposed a novel organization for vector registers that allows them to be very large and still have low latency.

Sites⁹⁰ discussed the relative advantages and disadvantages of caches and registers in the context of on-chip local storage for microprocessors (although he also discussed the requirements for supercomputers). He claimed that the stale data problem is probably the hardest design problem in systems that make copies of data (this is even true for single-processor designs). He suggested renaming memory locations as a hint to caching strategies, and advocated a top-of-stack cache using a "dribble-back" algorithm for memory updating. He noted that, ideally, data should be fetched from memory just far enough in advance that they arrive when needed.

Ditzel and McLellan²⁸ described a stack cache where the top of the stack is allocated to a small local memory. General addresses accessing the top of stack are mapped to stack-cache addresses when they are loaded into the instruction cache. They do not consider the tradeoff between latency and bandwidth of access.

2.5. Instruction Issuing and Instruction Formats

The work in instruction issuing mechanisms and instruction formats has concentrated on increasing the rate at which operations can be started by buffering a small number of instructions in high speed memory, by allowing instructions to issue out of program order, and by specifying operations more efficiently.

Strategies for issuing instructions have been investigated by many researchers. Takahashi, et.al.⁹⁷ described an instruction stack of 60 instructions implemented in tunnel-diode memory and running at CPU speed.

Thornton^{99,100} described the eight-word instruction stack in the CDC 6600. Up to four instructions can be stored in each word, so 32 instructions are held in the stack. Berlekamp⁸ reported that the instruction-stack loading algorithm of the CDC 6600 limited loops to 27 instructions or less. Thornton described the scoreboarding mechanism that allows later instructions to begin executing, even if earlier instructions are stalled and awaiting operands.

Tomasulo, et.al.¹⁰³ described the reservation stations, operand tags, and common data bus that allow the IBM System/360 Model 91 to execute instructions out of order if earlier instructions are stalled because of data dependency conflicts. The reservation stations allow instructions to await operands without keeping the functional units busy, so instructions on the Model 91 can start in cases where instructions on the CDC 6600 would stall.

Tjaden and Flynn¹⁰¹ discussed a simulation of the IBM 7090 with an enhanced instruction-issuing mechanism that allows parallel issuing of instructions. They reported a potential program speedup of 1.86. Riseman and Foster⁸⁵ described simulations of the CDC 3600 with an infinite instruction stack and parallel issuing of instructions. They reported an average program speedup of 1.76 when conditional branches inhibit parallel issuing. Tjaden and Flynn¹⁰² described a dynamic dependency-checking scheme using ordering matrices to represent program concurrency. Each instruction is described by a pair of binary vectors that completely describe the sources and sinks specified by the instruction, and tasks are represented as square matrices of dependency relationships. By using linear-algebraic-like operations, the concurrency can be exposed. Keller⁵¹ surveyed instruction issuing mechanisms for existing computers, and described a generalization of the Tomasulo algorithm that does not rely on a single common data bus. Rau⁸¹ examined the instruction buffers and instruction stacks of several high speed computers, concluding (among other things) that prefetch of instructions performs better than block fetches of instructions.

Weiss and Smith¹⁰⁰ reported on simulations of several instruction issuing mechanisms, including those of the Cray-1, the CDC 6600, and the Tomasulo algorithm. They reported that for scalar compilations of Livermore Kernels 1-14, the Tomasulo algorithm gives a total speedup of 1.58 over the Cray-1 algorithm, the CDC 6600 algorithm gives a speedup of 1.28 over the Cray-1 algorithm, and their own algorithm gives a speedup of 1.38 over the Cray-1 algorithm. They noted that the better algorithms require more hardware and could result in

a slower issue rate. Their work is flawed, however, because all their examples could have been scheduled statically, requiring no run-time scheduling hardware.

Patt, et.al.⁷⁶ described a microarchitecture that uses a generalization of the Tomasulo algorithm to execute a window of instructions as concurrently as data dependencies allow. This is useful when instruction timing cannot be predicted at compile time. This microarchitecture is intended to implement a number of dissimilar ISP architectures.

Maurer⁶⁶ presented a theoretical basis for instruction sets, but provided no insight into instruction set design. Tjaden and Flynn¹⁰¹ advocated adding explicit dependency tags to the instruction format to facilitate checking for dependency. They reported an 86% performance improvement when the IBM 7094 is so modified, without compiler techniques or programmer assistance. In light of the current state of compiler optimization and scheduling, their approach is not cost effective. Smith, et.al.^{92,93} described a simulated architecture with separate memory access instructions and execution instructions. They reported faster execution for their architecture than for the Cray-1, at least for scalar loops.

Flynn, et.al.³⁶ advocated complex instruction formats that better match the parse trees generated by compilers. They cited as advantages fewer redundant temporary registers and denser code. However, they required as many as 2500 opcodes—orders of magnitude more than conventional instruction formats require—and their instructions were harder to decode.

Flynn, Mitchell, and Mulder³⁷ reported on studies comparing the instruction-stream bandwidths and data-stream bandwidths of versions of a computer with different numbers of registers, caches, and instruction formats. They concluded that by adding a 16-bit register format and a register-memory format to a computer with a fixed 32-bit register-register format, the instruction-stream bandwidth requirements are reduced by one third to one half. They also reported that increasing the complexity of the instruction formats reduces the total memory-bandwidth requirements more than does increasing the number of registers.

Minimal-redundancy coding of data streams was discussed by Huffman in his landmark paper on the subject.⁴⁸

2.6. Other Factors

The design of any large system tends to be influenced by factors beyond the technical. Thornton¹⁰⁰ discussed many of the design decisions for the CDC 6600 as well as some of the mistakes made, including a logic simplification error that caused the pass instruction to initiate a divide (this error was corrected). Lincoln⁸³ discussed many of the human issues of a large project, including the effects of egos, high stress, and the murder of project leaders.* Swensen⁹⁵ discussed problems peculiar to development in a university environment.

Polya⁷⁸ discussed the process of problem solving in general in his delightful book. He suggested first understanding the problem to make sure it can be solved, devising a plan, perhaps using the solutions of similar problems, carrying out a plan with checks at each step, and checking the result for correctness and to see if it can be used to help solve other problems. Huff⁴⁷ listed many pitfalls of statistical analyses, including sampling biases, different methods of calculating averages, and statistical error. His advice is particularly well-suited for reporting the results of computer performance studies.

2.7. Past and Present Supercomputers

The evolution of supercomputers from the CDC 6600 and the IBM System 360 Model 91 to the Cray X-MP, ETA-10, and the NEC SX-2 has involved increases in clock speed, memory capacity, and the addition of vector instructions.

* A deranged ex-employee shot and killed the designer of the Floating Point unit of an experimental CDC STAR project. The incident was so upsetting that for several months the managers were unable to deal with the work he had left behind.

The CDC 6600 is described in detail in Thornton's book on that machine,¹⁰⁰ and a brief description is also found in his paper.⁹⁹ The CDC 6600 featured multiple functional units that supported overlapped execution, ten peripheral processing units to handle input and output tasks, a set of 24 registers, and a 100-nanosecond clock period. Berlekamp^{8,9} described the machine from the point of making it run as fast as possible, and discussed weak points in the design, such as blocking of loads and stores in order to prevent pathological memory hazards and the effective shortening of the instruction stack from eight to seven words.

The CDC 7600 bears a strong resemblance to its predecessor, the CDC 6600, so Thornton's book is still relevant. The CDC 7600 reference manual¹⁸ describes the hardware in detail. The primary differences between the CDC 6600 and the CDC 7600 are the fully pipelined functional units on the CDC 7600, the availability of up to five additional peripheral processors, and the shortening of the clock period to 27.5 nanoseconds.

The IBM System/360 Model 91 is described primarily in three articles published at the time of its announcement, those of D. Anderson, et.al.,³ S. Anderson, et.al.,⁴ and Tomasulo.¹⁰³ Kogge also has a fairly extensive discussion of this machine in his text.⁵² This computer had a 60 nanosecond clock period and supported data forwarding and hardware reservation stations that allowed out-of-order execution of instructions.

The CDC Star-100 was one of the first vector computers, storing its vectors in main memory and streaming them into pipelined functional units. Ramamoorthy and Li described the CDC Star-100 in their survey,⁸⁰ as did Kogge in his text.⁵² Lincoln⁶⁴ discussed some of the reasons for design choices made in that machine. Lambiotte and Voight⁶⁰ considered the Star-100 from the perspective of algorithm optimization.

The Texas Instruments ASC is described in a paper by Watson,¹⁰⁷ as well as in Ramamoorthy and Li's survey.⁸⁰ An excellent comparison of the TI ASC and the CDC Star-100 is included in this survey. The main features of the TI ASC were its vector instructions,

vector storage in memory and up to four homogeneous multifunctional units.

Russell described the architecture and implementation of the Cray-1,⁸⁸ and the details of the hardware implementation were discussed by Kolodzey.⁵⁴ Many details of the architecture are described in the hardware reference manual.²¹ Kogge⁵² also covered the Cray-1 in his text, and Bucher presented extensive measurements of the Cray-1 in her paper.¹⁴ The Cray-1 is analyzed in more detail in Chapter 6 of this dissertation.

The Cray X-MP is described in a paper by Larson,⁶² as well as in its hardware reference manual,²² and is also analyzed in Chapter 6 of this dissertation. Dongarra³⁰ reported on measurements of the Cray X-MP and other supercomputers. The Cray-2 is described in its hardware reference manual,²³ and is analyzed in Chapter 6 of this dissertation.

Lincoln described the evolution of the Cyber 205 from the Star-100,⁶⁴ and gave reasons for many design choices. Bucher¹⁴ described measurements of the Cyber 205 for benchmark programs at Los Alamos National Laboratories. The ETA-10,^{33,34,84} a descendent of the Cyber 205, is described in its hardware reference manual and is analyzed in Chapter 6. The ETA-10 represents a very different approach to supercomputing than the other supercomputers discussed, in that it has complex instructions and memory-to-memory vector instructions.

The HEP-1²⁵ is described in its hardware reference manual, and is analyzed in Chapter 6. It has thousands of registers, multiple pipelined functional units, and hardware support for multiple instruction streams, but it also has an incongruously slow 100-nanosecond clock period.

The NEC SX-2^{40,49,104,108,111} is faster than the Cray-1, the Cray X-MP and the Cray-2, for some applications. It is analyzed in detail in Chapter 6 of this dissertation. Among its features are a six-nanosecond clock, multiple copies of pipelined functional units, and hundreds of registers.

CHAPTER 3

Program Characteristics

The hardware requirements for fast execution depend on the characteristics of the programs to be executed. For example, the extent to which various operations should be supported depends on the relative frequencies of the various operations, and the support for parallel execution depends on the degree of program parallelism. The temporary storage requirements also depend on the characteristics of the programs.

Scientific programs are of interest in this thesis. Livermore Kernels 1-14⁶⁷ are chosen as the representative benchmarks because of their conciseness and widespread availability.

The programs are analyzed at the level of their dependency graphs. This eliminates anomalies introduced by specific compilations for specific machines, as well as anomalies imposed by specific programming languages. Exotic parallelization techniques such as factoring of expressions, as described by Kuck,⁵⁵ are not performed on the programs or dependency graphs.

The analysis technique reported here partitions the dependency graphs into sets of operations that can execute independent of each other. These partitions are then analyzed as to size, distributions of operations, and other characteristics. Any two programs with the same partition structure should have the same hardware requirements and execution times.

The execution times of programs are predicted from the partition structures. Upper and lower bounds for execution times are established for varying operation-execution times and degrees of parallel execution, but ignore implementation-dependent hardware resource conflicts.

The analyses and predictions are then tested by simulating execution, varying the relevant parameters, such as operation-execution times, amount of parallel execution allowed, and numbers and speeds of temporary storage locations. Arbitrary machine-specific restrictions on execution are avoided, although only simple operation scheduling is performed.

Based upon the results of the analyses and simulations, conclusions are drawn about the hardware support necessary, with an emphasis on temporary storage requirements.

3.1. Livermore Kernel Benchmark Descriptions

The Livermore Kernels⁶⁷ are a collection of small program kernels intended to represent the workload at Lawrence Livermore National Laboratories. The iteration counts for the loops range from 20 to 1024. Most operations in these kernels are floating-point operations, except for some integer or logical operations performed in kernels 13 and 14 in conjunction with array indexing. The kernels are listed in Appendix B, with a summary listed in table 3.1.

Table 3.1: Livermore Kernels 1-14, with Iteration Counts and Operation Counts.

Kernel	Title	Iterations	Operations
1	Hydro Excerpt	400	3204
2	MLR, Inner Product	40	800
3	Inner Product	1024	4098
4	Banded Linear Equations	128	1542
5	Tri-Diagonal Elimination, Below Diagonal	997	4981
6	Tri-Diagonal Elimination, Above Diagonal	997	4986
7	Equation of State Excerpt	120	2408
8	PDE Integration	20	2062
9	Integrate Predictors	100	2808
10	Difference Predictors	100	2900
11	First Sum	999	2998
12	First Difference	1000	3001
13	2-D Particle Pusher	128	3456
14	1-D Particle Pusher	150	3601

3.2. Dependency Graphs

Dependency graphs are directed acyclic graphs, with the nodes of the graphs representing operations and the arcs of the graphs representing data dependencies or control dependencies.

Each operation changes the architectural state of the machine; that is, the change is visible to the assembly-language programmer. For example, a multiply or add writes a register with a product or sum of two other registers, a load writes a register with the contents of a memory location, and a branch writes a program counter with a program address.

Several operations can start simultaneously on some computers. For example, when k Cray-1-style vector instructions are running overlapped, k operations start each clock tick (one from each vector instruction).²¹⁻²³ A single instruction can also start several operations simultaneously. For example, NEC SX-2 vector instructions start four operations each clock tick, so k NEC SX-2-style vector instructions running overlapped could start $4k$ operations each clock tick.⁴⁰

As discussed in Chapter 1, an instruction is the specification of one or more operations to be performed. Instructions are only considered peripherally in these analyses.

An operation *starts* or begins execution when some subset of its required operands and hardware resources become available. For these analyses, all operands and resources must be available before an operation can start.

An operation *finishes* when it completes execution and writes its result, making its last change to the architectural state of the machine.

Each operation is performed as a collection of micro-operations, each of which may or may not change the architectural state of the machine. For example, a store is performed as three micro-operations:

- (1) a relative address is read from a register (an address register or the instruction register);
- (2) a datum is read from an architectural register;
- (3) the datum is written into memory at the physical address.

Micro-operations (1) and (2) can execute in parallel, while micro-operation (3) must wait for both (1) and (2) to finish before it can execute.

3.2.1. Dependency-Graph Generation

Dependency graphs of programs are generated semi-automatically, using a degree of sophistication similar to that of current vectorizing compilers.

Loops executed n times are unrolled so that all n iterations can execute concurrently, if data dependencies allow them to do so. The loop test operations are not included in the dependency graph if n is known at compile time. Scalar temporaries are duplicated, if necessary, to allow several iterations to run concurrently. Intermediate values that are not available outside loops are not maintained or stored, so machine states after exceptions may not be consistent with serial execution models. (Of course, an actual compilation would need to include code to guarantee a consistent state following an exception.)

Simple code transformations are performed, particularly those involving application of associative properties of arithmetic. For example, the loop

```
for i = 1 to n
  q = q + x[i]
```

is transformed to the statement

$$q = q + \sum_{i=1}^n x[i],$$

and the sum is performed using a binary summation tree. This transformation allows speed-ups with parallel hardware, yet, since it adds no extra operations to the original program, it does not slow down execution with serial hardware (it *does* require more temporaries,

however, but these do not necessarily slow the overall execution rate).

More complex transformations, such as the parallelizations of tri-diagonal elimination described by Lambiotte and Voight,⁶⁰ increase parallelism but add extra operations, and would, thus, slow execution with serial hardware. Although these complex transformations could be performed, they are not performed during dependency graph generation for these analyses. If they were performed, more program parallelism might be exposed.

The degree of sophistication during dependency graph generation is roughly equivalent to that of current vectorizing compilers, such as Cray Research's CIVIC, version 131e.²⁴ It represents a lower bound on parallelism that can be extracted using parallelizing or vectorizing compilers.

3.3. Program Partitioning

Programs in the form of a dependency graph are partitioned in the following manner.

- (1) An eager partitioning of the dependency graph is formed such that:
 - (a) partition 0 contains all operations that depend on no operation;
 - (b) for all partitions from 1 to $p-1$ partition i contains all operations that depend only on operations in partitions 0 to $i-1$, and that depend on at least one operation in partition $i-1$.

The eager partitioning represents the execution order of operations on a parallel computer with unlimited resources and uniform operation-execution times with an eager execution strategy.

- (2) A lazy partitioning of the dependency graph is formed such that:
 - (a) partition $p-1$ contains all operations that source no operations;

- (b) for all partitions from $p-2$ to 0, partition i contains all operations that source only operations in partitions $p-1$ to $i+1$ and that source at least one operation in partition $i+1$.

The lazy partitioning represents the execution order of operations on a parallel computer with unlimited resources and uniform operation-execution times with a lazy execution strategy.

- (3) The critical path of the dependency graph is formed, and is defined as the intersection of eager partition 0 with lazy partition 0, eager partition 1 with lazy partition 1, and so on.

The critical path contains elements from all p partitions.

- (4) A scheduled partitioning of the dependency graph is formed by allocating operations to partitions such that:
 - (a) stores are executed as eagerly as possible;
 - (b) operations with more sources than sinks are executed as eagerly as possible;
 - (c) all other operations are executed as lazily as possible.

This partitioning tends to reduce the lifetimes of intermediate results. Operations in the critical path have the same partitions for all partitioning methods, but some non-critical-path operations may have different partitions than they did with either eager partitioning or lazy partitioning.

Other scheduled partitionings are also possible; for example, operations not in the critical path could be allocated to partitions such that partition sizes are as uniform as possible.

3.3.1. Partitioning Example

The partitioning process is illustrated in the following example, based on Livermore Kernel 6 with $n = 3$.

```

for i = 2 downto 1
  x[i] = x[i] - z[i] * x[i+1]

```

There are 11 operations: five loads from arrays, two multiplies, two subtracts, and two stores to an array. The dependency graph for this program is shown in figure 3.1.

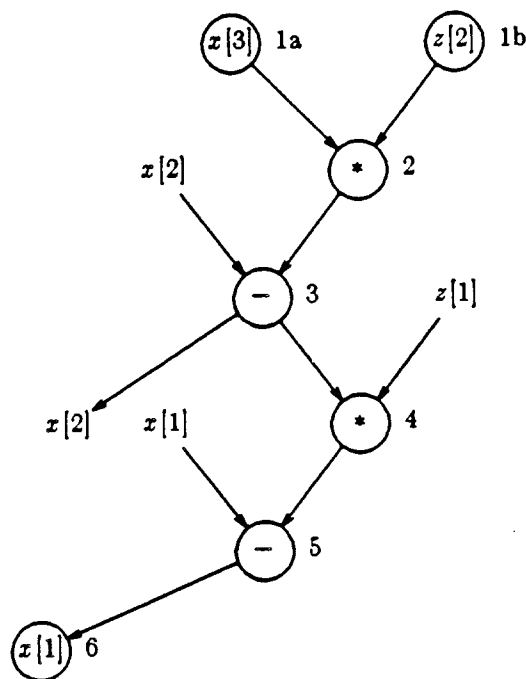


Figure 3.1: Dependency Graph for Livermore Kernel 6 with $n = 3$.

This dependency graph has a critical-path length of six operations; the six operations are circled and numbered. The dependency graph can be partitioned into six sets of independent operations; each partition contains one or more operations that are in the critical path. The eager partitioning, the lazy partitioning, the critical path, and a scheduled partitioning are shown in table 3.2. "L" indicates an array load, "*" indicates a multiply, "-" indicates a subtract, and "S" indicates an array store.

With the eager partitioning, all five array loads can start immediately, but the multiplies, subtracts, and stores depend on operations in earlier partitions.

Table 3.2: Eager, Lazy, and Scheduled Partitioning and Critical Path for the Example in Figure 3.1.

Partition	Partitioning Method			
	Eager	Lazy	Critical Path	Scheduled
0	5L	2L	2L	2L
1	1*	1*, 1L	1*	1*, 1L
2	1-	1-, 1L	1-	1-, 1L
3	1*, 1S	1*, 1L	1*	1*, 1L, 1S
4	1-	1-	1-	1-
5	1S	2S	1S	1S

With the lazy partitioning, three of the loads are moved to the partitions just before they are used, and the second store is moved down to the last partition.

The intersection of the eager and lazy partitioning is the critical path of the program: the two initial loads, the multiplies and subtracts, and the last store.

In this scheduled partitioning of the program, all operations except stores are executed lazily; stores are executed eagerly.

This analysis shows that, for programs with this dependency graph structure and scheduled partitioning, at most three operations can be running at a time, that at most two memory operations can be in progress at one time, and that multiplies and subtracts do not overlap. This analysis can be used to make architectural decisions; parallel or pipelined arithmetic functional units would be under-utilized for programs like this, as would be more than two memory ports. Furthermore, no more than two registers would be needed to hold temporary results.

3.3.2. Prediction of Execution Times from Partitions

Theoretical upper and lower bounds on the execution times of programs can be established. These assume no hardware conflicts, although there is a limit on the maximum number of operations that can start simultaneously. The predictions are thus independent of the anomalies of particular hardware configurations, and reflect the nature of the programs themselves. Hardware requirements for conflict-free execution can be determined later by examining the distributions of operations in the partitions.

Definition: *Start-limit* is the maximum number of operations that can be started each clock tick. Typically, a *start-limit* that is greater than one is achieved by overlapping the execution of several vector instructions, each of which start one operation every clock tick.

Definition: E_{time_j} is the execution time of operation j .

Definition: O_{pcount_i} is the number of operations in partition i .

Definition: $First_i$ is the first operation started in partition i .

Definition: $Earliest_i$ is the first operation in partition i to finish. $Earliest_i$ could be different than $first_i$ if $earliest_i$ had a shorter execution time than $first_i$.

Definition: $Last_i$ is the last operation started in partition i .

Definition: $Latest_i$ is the last operation in partition i to finish. $Latest_i$ could be different than $last_i$ if $latest_i$ had a longer execution time than $last_i$.

Lemma 3.1: With eager partitioning, some operation in partition i must finish before the first operation in partition $i+1$ can start.

Proof: Every operation in partition $i+1$ depends on at least one operation in partition i (by the definition of eager partitioning), so it must wait for that operation to finish before that operation's result is available as an operand. ■

Lemma 3.2: With lazy partitioning, some operation in partition i cannot start until all operations in partition $i-1$ finish.

Proof: Every operation in partition $i-1$ sources at least one operation in partition i (by the definition of lazy partitioning), so an operation in partition i must wait for $latest_{i-1}$ to finish before it can start. ■

Lemma 3.3: The execution time of all operations in partition i is at least

$$\frac{opcount_i}{start-limit} + etime_{last_i} \text{ ticks.}$$

Proof: Assume all operations that source operations in partition i have finished by the time the first operation in partition i starts. All operations in partition i must still start before they can finish, and that requires

$$\frac{opcount_i}{start-limit}$$

clock ticks. The latest operation must finish, and that happens no sooner than when the last operation started finishes, which takes an additional $etime_{last_i}$ ticks. If some operations sourcing operations in partition i had not finished, this could only delay the execution of operations in partition i and increase the execution time. ■

Lemma 3.4: The execution time of all operations in all p partition is at least

$$\left[\frac{\sum_{i=1}^p opcount_i}{start-limit} \right] + etime_{last_p} \text{ ticks.}$$

Proof: Assume that the execution time $etime_{latest}$, of the last operation in partition i can be overlapped with the starting of operations in partitions $i+1$ through p , $1 \leq i < p$, and that operations from two adjacent partitions can be started during the same clock tick if $start-limit \geq 2$. Then, only the execution time of the last operation in partition p cannot be overlapped with later operations, and in the best case, only the start times of the last operations in partition p cannot be overlapped with operations from adjacent partitions. If this assumption is not true for all $1 \leq i < p$, then less overlap occurs, and the total execution time is greater. ■

Lemma 3.5: Assuming all operations in partitions before i have finished, the execution time of all operations in partition i is at most

$$\frac{opcount_i}{start-limit} + etime_{latest} \text{ ticks.}$$

Proof: All operations in partition i are independent and can execute concurrently (by the definition of partitioning), but they must all start before they can finish, and that requires

$$\frac{opcount_i}{start-limit}$$

clock ticks. The latest operation must finish, and that happens no later than $etime_{latest}$ ticks after the last operation in partition i starts. ■

Theorem 3.6: The execution time of all operations in all p partition is at most

$$\left[\sum_{i=1}^p \left(\frac{opcount_i}{start-limit} + etime_{latest} \right) \right] \text{ ticks.}$$

Proof: Assume that $first_{i+1}$ depends on all operations in partition i , for $1 \leq i < p$, so $first_{i+1}$ cannot start until $latest_i$ finishes. From Lemma 3.5, this is no more than

$$\frac{opcount_i}{start-limit} + time_{latest_i} \text{ ticks.}$$

Executing all operations in partition p requires an additional

$$\frac{opcount_p}{start-limit} + time_{latest_p} \text{ ticks.}$$

If the assumption is not true, some execution can be overlapped, and the total execution time is less. ■

The pessimistic model of execution assumed in theorem 3.6 allows no overlap of critical-path-operation execution and operation starting. A more optimistic model of execution allows the critical-path-operations from each partition to start first, so their execution can run in parallel with the starting of other operations.

The minimum number of operations that must start each partition before operations in the next partition can start are given by the lazy partitioning of the dependency graph. The cumulative minimum number of operations started in each partition is just the sum of lazy partition sizes.

Definition: $Minstarts_i = \sum_{j=1}^i lazy_j.$

The maximum number of operations in each partition that have all their operands and can start are given by the eager partitioning of the dependency graph. The cumulative maximum number of operations started in each partition is just the sum of eager partition sizes.

Definition: $Mazstarts_i = \sum_{j=1}^i eager_j.$

While the *earliest* operation in a partition executes, several other operations in the partition may be able to start without additional time penalty. The allowable overlap of starts and execution is determined by the execution time of the *earliest* operation in each partition and by the *start-limit* of the computer being modeled.

Definition: $Overlap_i = etime_{earliest} * start-limit.$

The cumulative number of operations started is the sum of operations started each partition; this number must be between $minstarts$ and $mazstarts$. Subject to these constraints, $starts_i$ is the sum of $starts_{i-1}$ and $overlap_i$.

Definition: $Starts_0 = 0$

Definition: $Starts_i = \min(mazstarts_i, \max(starts_{i-1} + overlap_i, minstarts_i)).$

Under this model of execution, the time spent in each partition is either the time to execute the *earliest* operation in the partition, or the time to start enough operations to reach $minstarts$ for this partition, which ever is greater.

Definition: $Time_i = \max \left(etime_{earliest}, \frac{minstarts_i - starts_{i-1}}{start-limit} \right)$

Theorem 3.7: The execution time of all operations in p partitions is at least

$$\sum_{i=1}^{p-1} time_i + \frac{minstarts_p - starts_{p-1}}{start-limit} + latest_{p-1}$$

Proof: In the best case, for i between 1 and $p-1$, an operation in partition $i+1$ can start after the first operation in partition i finishes, (using Lemma 3.1), or after enough operations start so that $starts_i \geq minstarts_i$. If $overlap_i$ is large enough and if $mazstarts_i$ is large enough, extra operations can be started in partition i , reducing the number of operations that must be started in later partitions. After they all start, the operations in partition p must still execute, and this execution time is specified by Lemma 3.3. ■

Remark: Note that $minstarts_i \leq starts_i \leq mazstarts_i$, by definition. Then

$$\max \left(ctime_{earliest}, \frac{minstarts_i - maxstarts_{i-1}}{start-limit} \right) \leq time_i, \text{ and}$$

$$time_i \leq \max \left(ctime_{earliest}, \frac{lazy_i}{start-limit} \right),$$

so this model of execution is at least as fast as a lazy model of execution.

3.3.3. Partition-Size Analysis

The kernels fall into one of two groups: parallel kernels and serial kernels. The parallel kernels 1, 2, 3, 4, 7, 8, 9, 10, and 12 have mean scheduled-partition sizes between 47 and 1868 operations, and have mean critical-path widths between 47 and 1099 operations. All are vectorized by the CIVIC compiler.²⁴ The serial kernels 5, 6, 11, 13, and 14 have mean scheduled-partition sizes between 2.5 and 8.9 operations, and have mean critical-path widths of 1.0. Kernels 5, 6, and 11 are not vectorized because of potential dependency conflicts, and kernels 13 and 14 are not vectorized because of indirect array indices.²⁴ Partition size data are presented in table 3.3.

Table 3.3: Partition Sizes and Critical-Path Widths.

Kernel	Partition Size			Critical-Path Width		
	Max.	Mean	Std. Dev.	Max.	Mean	Std. Dev.
1	800	534.00	206.05	800	467.17	163.06
2	320	133.33	106.33	320	113.33	111.47
3	2048	292.71	581.30	2048	292.64	581.34
4	768	140.18	238.81	768	139.91	238.98
5	3	2.50	0.50	2	1.00	0.02
6	3	2.50	0.50	2	1.00	0.02
7	360	200.67	77.71	121	120.08	0.29
8	720	294.57	216.53	360	173.14	93.43
9	800	401.14	295.53	707	329.57	276.75
10	300	263.64	67.42	200	109.09	30.15
11	3	3.00	0.08	2	1.00	0.03
12	1001	1000.33	0.58	1001	1000.33	0.58
13	257	8.86	21.94	2	1.02	0.12
14	9	5.94	2.25	3	1.01	0.10

The execution times of serial kernels, bounded by theorems 3.6, and 3.7, tend to be dominated by the execution times along the critical paths, rather than by the start times,

because typical execution times are larger than typical partition sizes for serial kernels. Conversely, the execution times of parallel kernels tend to be dominated by start times, rather than by execution times, because partition sizes are so much larger than execution times. For example, kernel 1 would need to start over 76 operations each clock tick before the time to start operations in each partition would be as small as a typical operation-execution time of seven clock ticks.

Thus, serial programs need low execution times for fast execution, while parallel programs need high start-limits for fast execution.

3.3.4. Memory-Reference Analysis

If most memory references are loads from addresses known at compile time, longer memory access times can be tolerated by starting loads well before they are needed. The analysis summarized in table 3.4 indicates that memory references with addresses that are data-independent account for from 22% to virtually all memory references.

Table 3.4: Fraction of Memory References with Data-Independent Addresses.

Kernel	Fraction of References
1	0.67
2	0.91
3	1.00
4	1.00
5	0.67
6	0.67
7	0.75
8	0.61
9	0.91
10	0.50
11	0.50
12	0.50
13	0.22
14	0.27

For kernels 13 and 14 the low fraction of independent memory references slow the execution on computers with high-latency memory. Thus, the Cray-2,²³ with memory twice as slow as

the Cray X-MP,²² is much worse suited for applications with this kind of memory reference characteristics.

Most memory references in these kernels are array references, suggesting that two types of memory references should be supported. A high-bandwidth, higher-latency array reference mechanism could be used when addresses are known ahead of time. When addresses are generated in the critical path, a lower-bandwidth, low-latency memory reference mechanism would be used. Ideally, any variable should be accessible by either type of memory reference, but this would be difficult to implement. More likely, separate scalar and vector memories would be used.

3.3.5. Temporary-Storage-Requirement Analysis

Storing temporary results in main memory wastes memory bandwidth and increases the execution time if the results are used in the critical path. Table 3.5 summarizes the analysis of result lifetimes.

Table 3.5: Result Lifetimes (in Partitions) for Short-Result-Life Scheduling and Eager-Execution Scheduling.

Kernel	Shortlife Heuristic			Eager Execution		
	Max.	Mean	Std. Dev.	Max.	Mean	Std. Dev.
1	1	1.00	0.00	4	1.29	0.70
2	1	1.00	0.00	3	1.11	0.45
3	1	1.00	0.00	12	1.00	0.17
4	1	1.00	0.00	9	1.02	0.35
5	1	1.00	0.00	1992	498.63	642.77
6	1	1.00	0.00	1994	499.13	643.42
7	10	1.58	2.03	10	1.91	2.25
8	3	1.13	0.50	3	1.20	0.54
9	1	1.00	0.00	3	1.11	0.42
10	1	1.00	0.00	9	2.89	2.67
11	1	1.00	0.00	999	250.38	322.28
12	1	1.00	0.00	1	1.00	0.00
13	385	19.82	66.23	384	19.72	65.98
14	595	1.63	10.37	598	28.51	100.39

Each result is usually used in the partition after its generation, especially with short-result-life

scheduling; however, some results are used up to 595 partitions after generation. The short lifetimes of so many results suggests that some temporary storage could be optimized for speed, density, etc., without regard for long-term data integrity. For example, capacitive storage devices without refresh mechanisms might be practical in some technologies. Alternatively, results might propagate through short delay lines before they are used, rather than waiting in registers.

An analysis of the number of times each result is used is summarized in table 3.6. Most results are only used once, although some are used as many as 720 times. Kernel 10 shows fewer than one mean use of each result because its stores are not used within the loop.

Table 3.6: Number of Times Each Result Used.

Kernel	Number of Uses		
	Max.	Mean	Std. Dev.
1	400	1.37	12.22
2	1	0.95	0.22
3	1	1.00	0.02
4	1	1.00	0.04
5	2	1.00	0.63
6	2	1.00	0.63
7	720	1.64	15.49
8	120	1.45	3.80
9	100	1.25	5.28
10	2	0.97	0.81
11	2	1.00	0.82
12	2	1.00	0.82
13	5	1.37	1.31
14	150	1.54	2.62

The frequency of results that are used once suggests that specialized mechanisms for transmitting results could be used. For example, operands could be specified implicitly or three-operand complex instructions (like $x * y + z$) could be used. Three-operand complex instructions could reduce by 20% the number of instructions required to perform the operations: load a pair of array elements, subtract the product of a scalar and one element from the other element, and store the difference in another array.

3.4. Execution Simulation

Execution of each of the kernels is simulated using the mechanism described here. Results are summarized in tables C.1-C.14 in Appendix C.

All operations have a count of the number of operands required. Operations that depend on no other operations have operand counts of zero.

A ready list contains operations that have all necessary operands and are ready to run. Initially, the ready list contains all operations that depend on no other operations.

Each clock tick, up to start-limit operations in the ready list are started and added to an active list of operations that are executing. Operations in the critical path are selected before operations not in the critical path, but no attempts are made to schedule operations more cleverly (clever scheduling heuristics might select operations based upon when dependent operations could start, for example). The execution time for the operation is included in the active list entry for the operation.

Each clock tick, the remaining execution time entry of each operation in the active list is decremented. When an operation's execution time reaches zero, the operation finishes, and it is removed from the active list.

Every operation that depends on a just-finished operation has its operand-count entry decremented, and when an operand-count entry reaches zero, the operation is added to the ready list.

A count is kept of the number of clock ticks from the time the first operation starts until the last operation finishes; this is the total execution time.

The variable parameters in these simulations are the start-limits and the execution times. Start-limits of one, two, four, and eight are used to determine the sensitivity of kernel execution times to start-limits. In general, the parallel kernels are more sensitive to the

start-limit than serial kernels, as predicted in section 3.3. The very-serial kernels 5, 6, and 11 are practically insensitive to the start-limit.

Execution times are taken from the Cray-1S operation-execution times,²¹ except that all hardware conflicts except for start-limits are ignored. When start-limits are increased, a constant cost is sometimes added to all operation-execution times to reflect greater operation-start-hardware complexity. This cost as modeled grows as the \log_2 of the start-limit. The simulations show that the serial kernels actually run slower when more operations start each clock tick, if starting more operations at a time increases their execution times. Uniform execution times of 1, 2, 4, 8, 16, 32, and 64 ticks per operation are also used. The simulations show that increasing the start-limit does not speed up execution very much, once the product of the start-limit and the operation-execution-time is as large as the mean partition size, as predicted in section 3.3.

The results of the simulations are presented in Appendix C, along with the execution-time bounds set by theorems 3.6 and 3.7. No schedule can achieve an execution time outside these bounds. Root-mean-square differences between bounds and simulation times are summarized in table 3.7. The differences show how closely the naive theoretical upper and lower bounds on execution time. The generally low differences between simulated execution times and lower bound predictions indicate that the naive scheduling is close to optimal for many different execution parameters. Differences between the predicted upper bounds and simulated execution times are higher, especially for serial programs. This is because the execution model assumed for theorem 3.6 allows no execution overlap between partitions, although some overlap is present in the simulations of each kernel. For parallel kernels, the execution time is dominated by the time to start all the operations in each large partition, so execution overlap is relatively unimportant. In contrast, the execution time of the serial kernels is dominated by operation-execution time, so any overlap of execution is significant.

Table 3.7: Root-Mean-Square Differences Between Execution Simulations and Bounds Set by Theorems 3.6 and 3.7, for Livermore Kernels 1–14.

Kernel	Bounds	
	Upper	Lower
1	12.8%	2.2%
2	21.4%	4.1%
3	7.7%	1.5%
4	12.1%	2.9%
5	55.9%	0.0%
6	55.9%	0.0%
7	29.5%	0.1%
8	19.6%	2.5%
9	14.6%	1.8%
10	26.1%	0.8%
11	63.2%	0.0%
12	6.9%	0.0%
13	62.9%	10.5%
14	67.8%	9.9%

3.4.1. Dependence of Performance on Number of Registers

The simulations discussed above assume an unlimited supply of fast registers for temporary results. The dependence of performance on the number of registers available is measured in the following manner.

The simulator is modified so that a finite number of registers is available. When any operation except a store finishes, it writes its result to either a register or to memory. If a register is available, the operation finishes execution after the normal number of clock ticks, the register is allocated, and the count of available registers is decremented. If no register is available, the result must be written out to memory. This extends the execution time by a store time, and also uses an additional operation's worth of start-bandwidth. This accounts for an extra store operation for a load-store architecture, or the addition of a full address specification on the instruction-stream bandwidth for a memory-to-memory or general-address architecture. Note that this register-allocation procedure is not optimal because it does not take into account how many times each result is used. However, the results summarized in

table 3.6 show that most results are only used once, so the chance of allocating the last available register to the wrong operation is small.

If an operation other than a load reads all operands from registers, it executes in the normal amount of time. If an operation must read one or more operands from memory, its execution time is extended by a load time, and it uses an additional operation's worth of start-bandwidth for every operand in memory.

After all sinks of a result in a register have started, the register is deallocated and made available for another operation's result.

Operations are scheduled to minimize the length of time that each result is allocated to a register, rather than to minimize running time, so in some cases the kernels execute more slowly than the other simulations. The results of the limited-register simulations are shown in figures 3.2a through 3.2n.

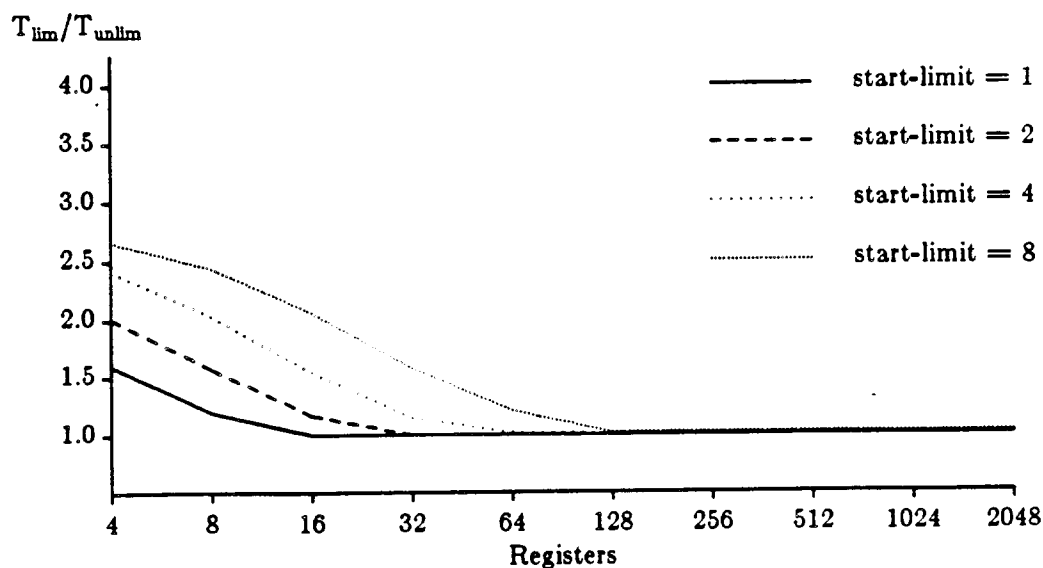


Figure 3.2a: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 1.

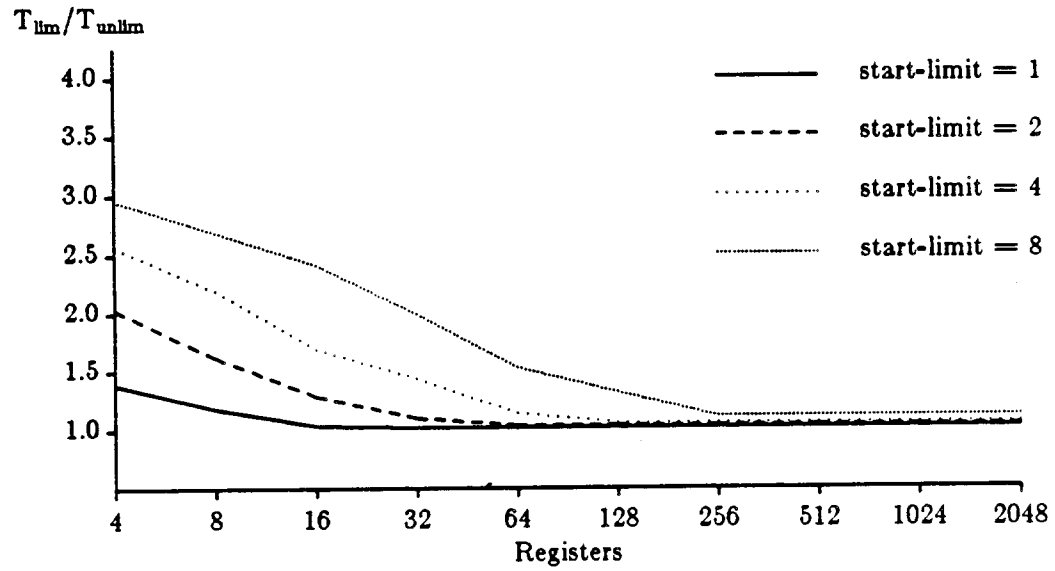


Figure 3.2b: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 2.

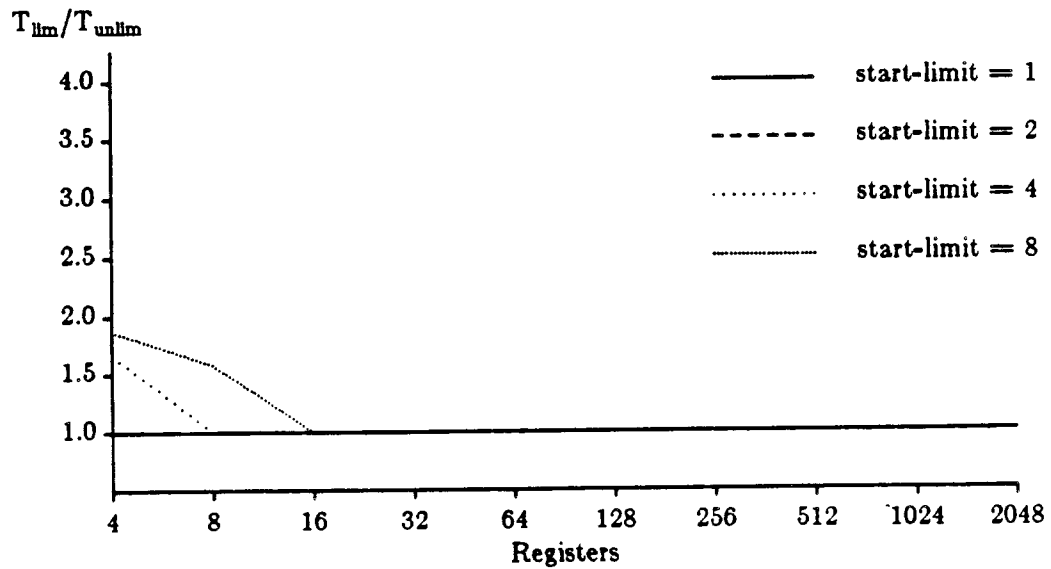


Figure 3.2c: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 3.

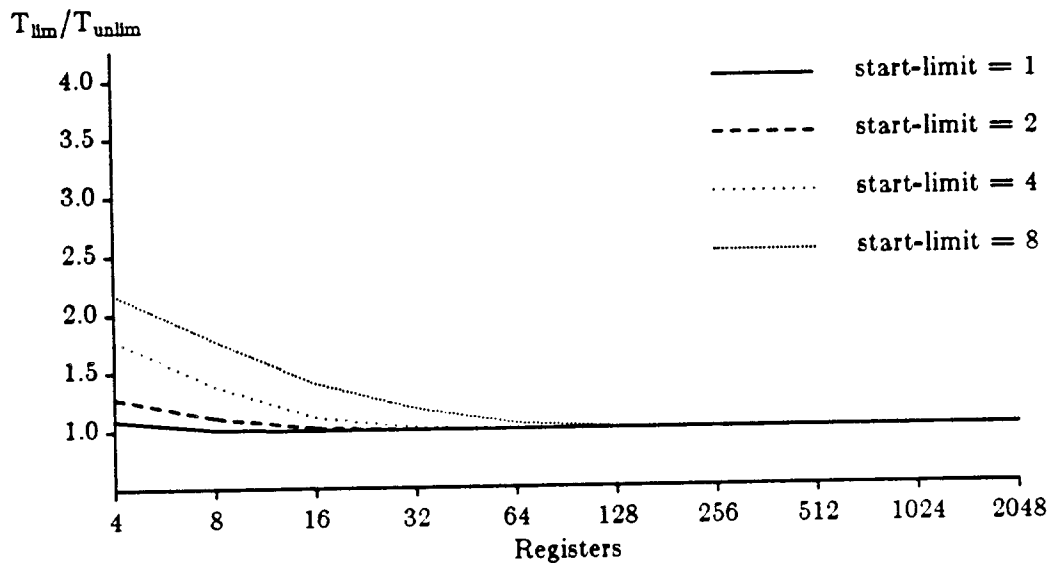


Figure 3.2d: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 4.

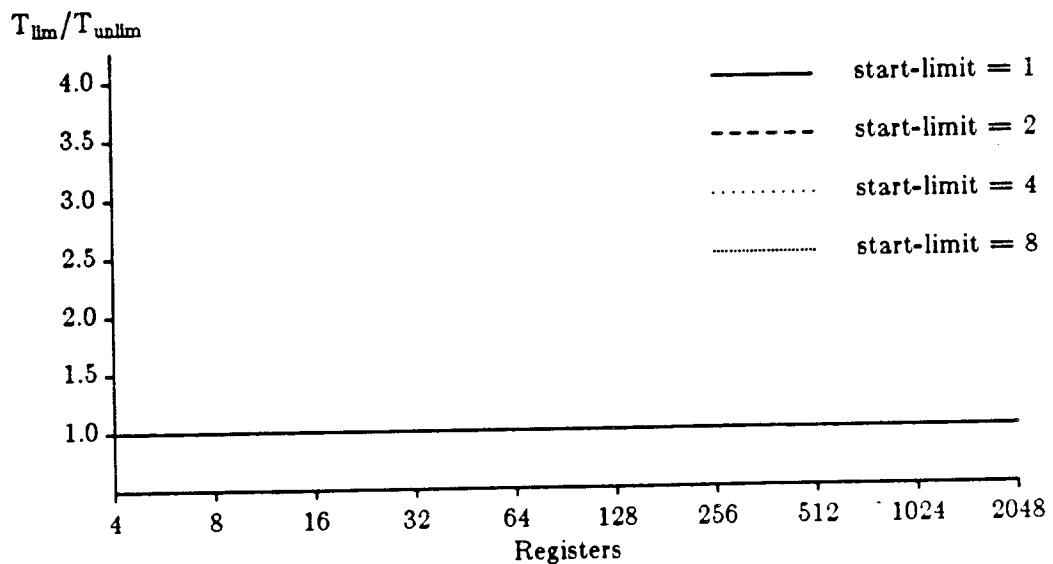


Figure 3.2e: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 5.

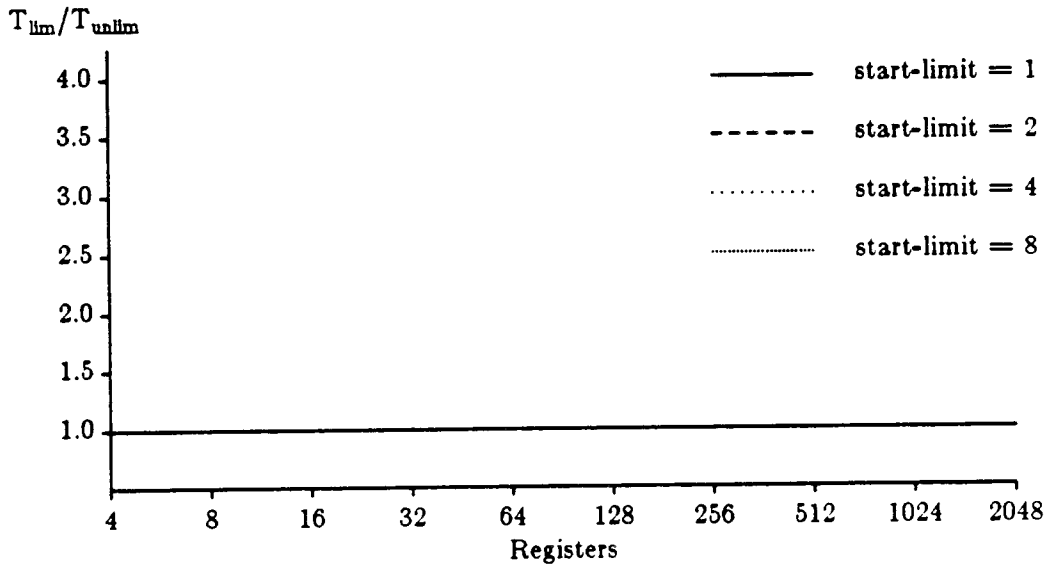


Figure 3.2f: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 6.

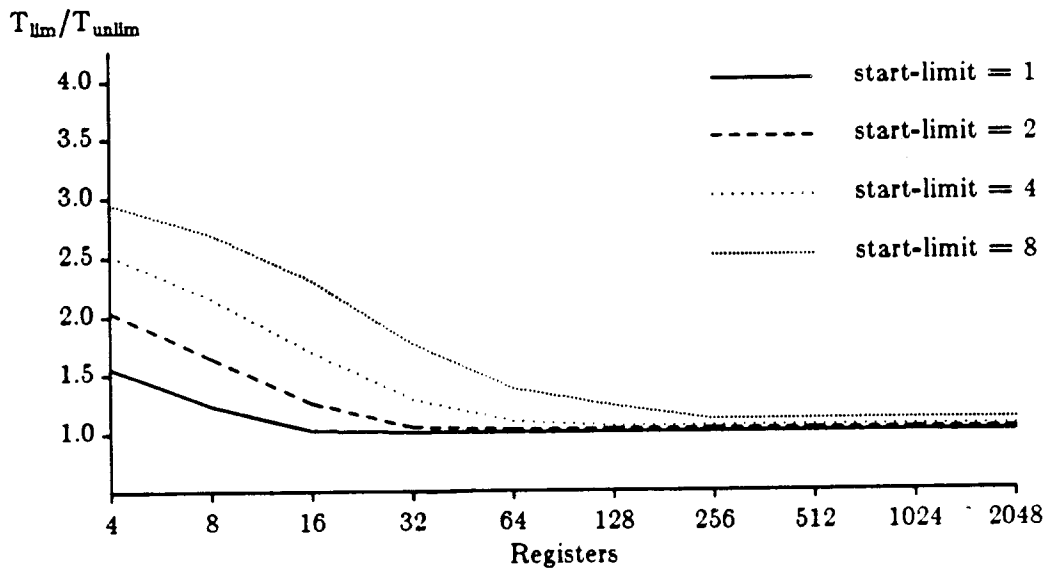


Figure 3.2g: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 7.

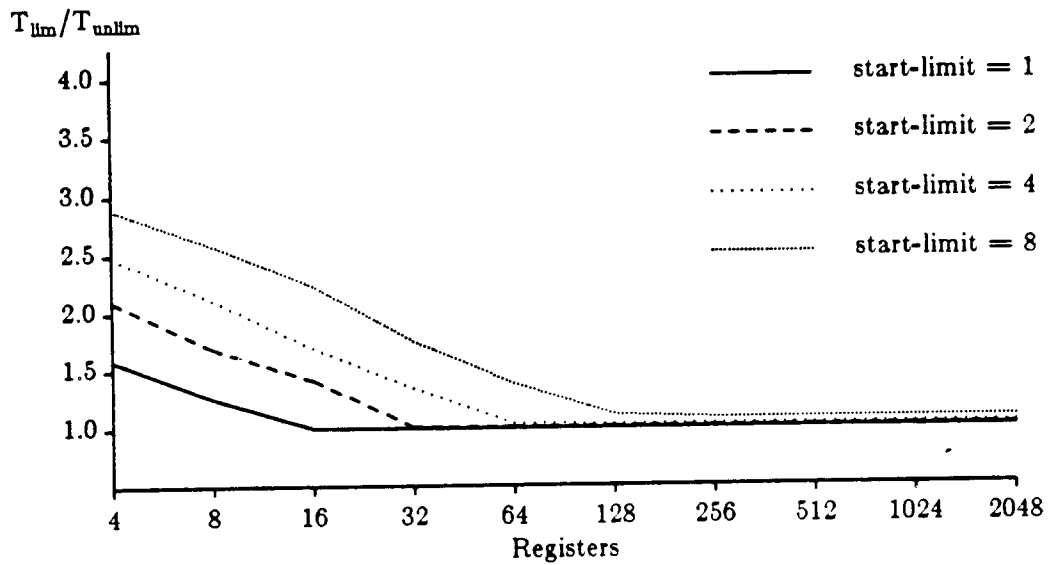


Figure 3.2h: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 8.

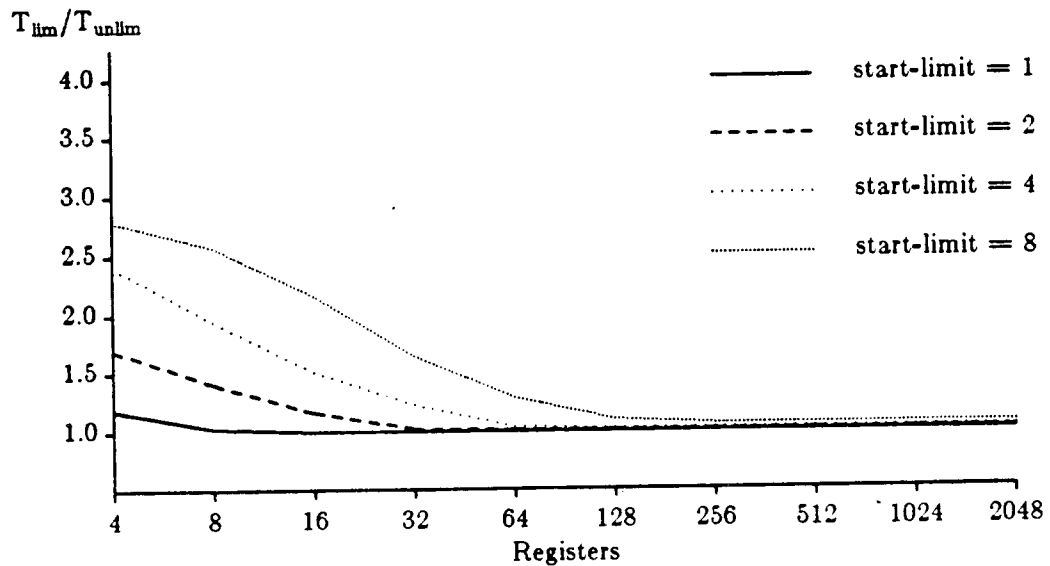


Figure 3.2i: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 9.

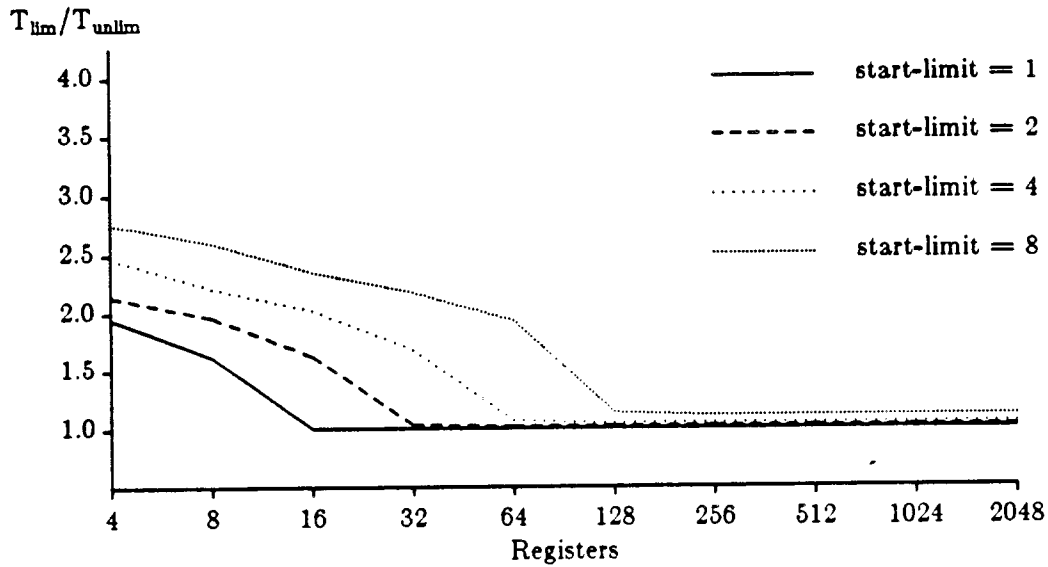


Figure 3.2j: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 10.

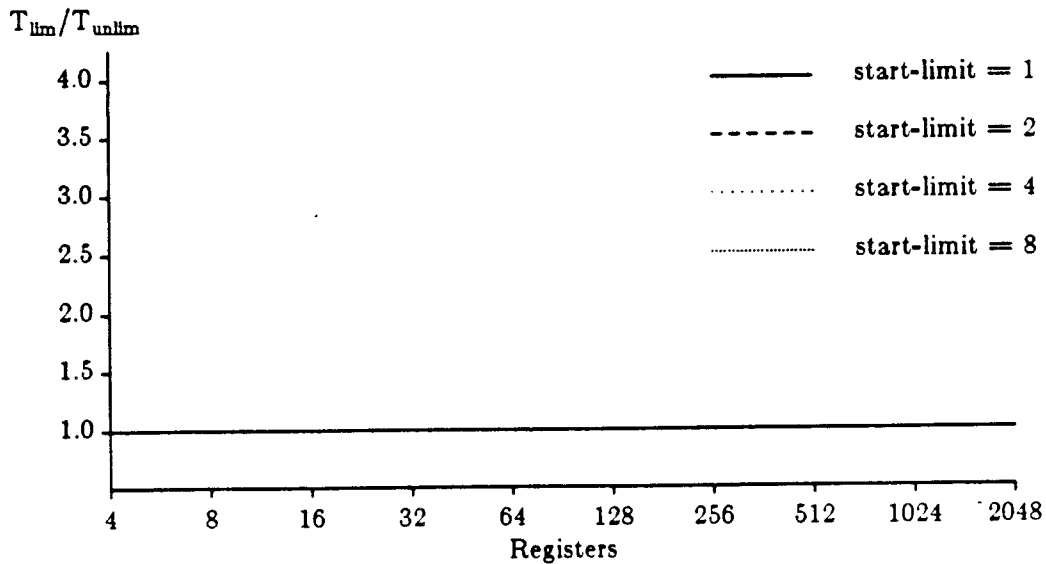


Figure 3.2k: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 11.

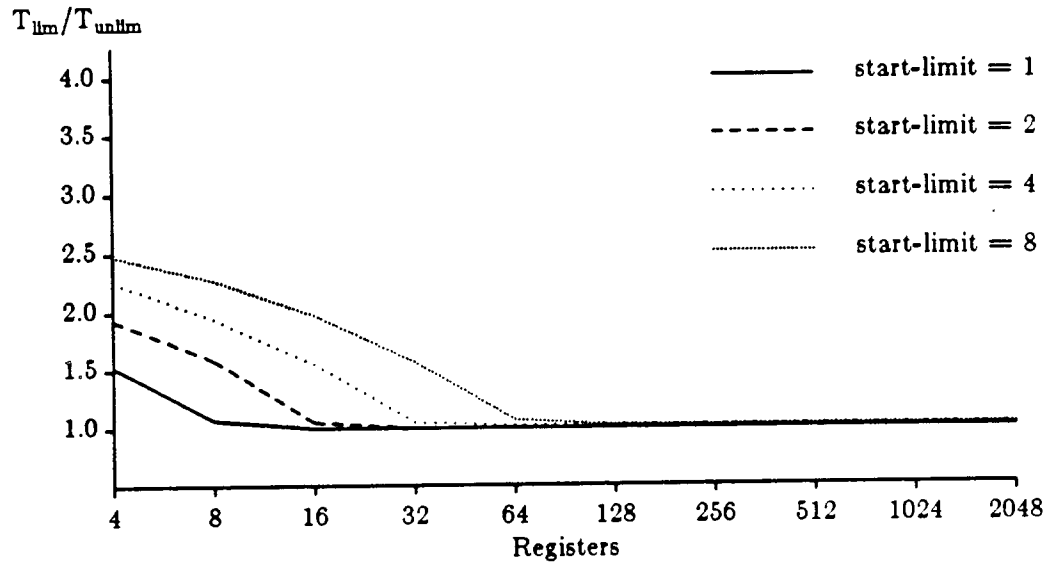


Figure 3.2l: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 12.

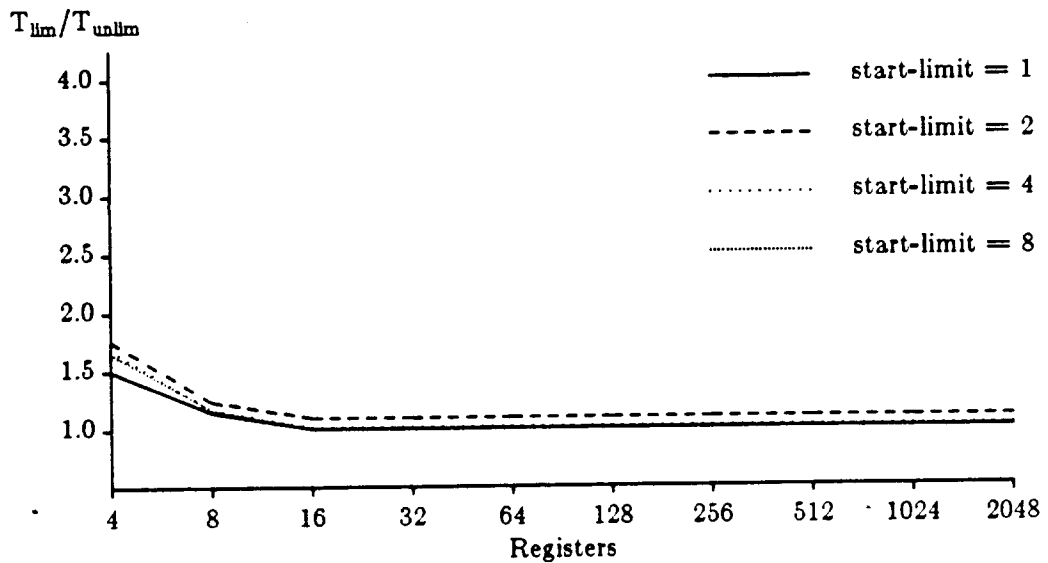


Figure 3.2m: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 13.

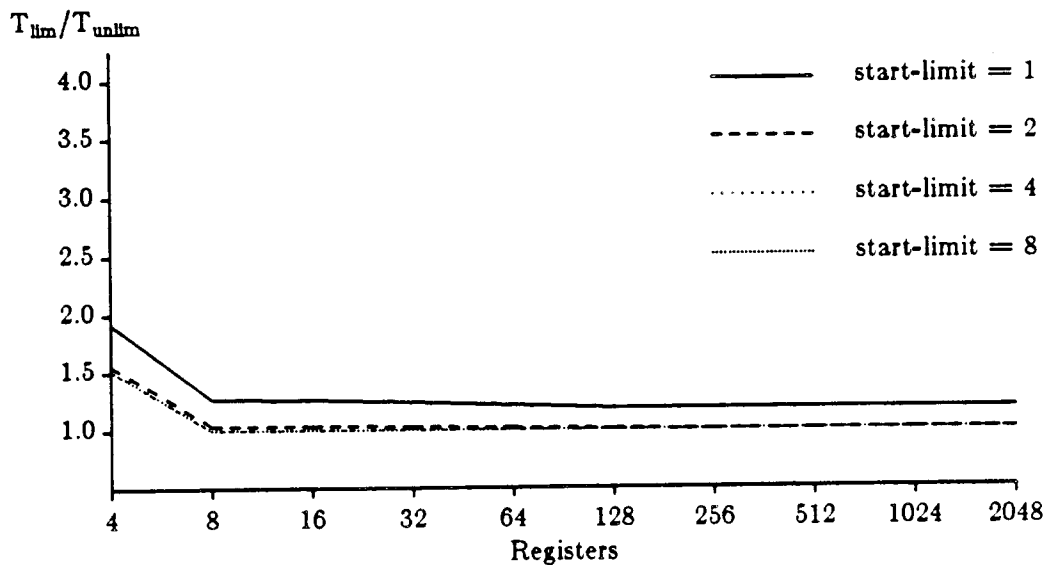


Figure 3.2n: Ratio of Execution Time with Limited Registers to Execution Time with Unlimited Registers, for Livermore Kernel 14.

The results do not vary significantly when the register-allocation heuristics are varied, suggesting that the allocation heuristics are reasonably good. The figures show the ratio of simulated execution time with n registers to the execution time with an unlimited number of registers, for n equal to 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 2048.

The very-serial kernels 5, 6, and 11 run as fast with four registers as they do with an unlimited number of registers. The narrow critical-path widths and small partitions-sizes of these kernels insure that few operations are executing concurrently, and that most results are used immediately after they are generated. The moderately-serial kernel 13 requires approximately sixteen registers to run at near to its ultimate speed, but the execution speed with the minimal-register schedule never reaches the speed of the kernel with a minimal-execution-time schedule that happens to require more registers.

The moderately serial kernel 14 requires approximately 128 registers to reach its ultimate speed for a start-limit of one, but this is 30% slower than the execution speed of the kernel with a minimal-execution-time schedule. Kernels 13 and 14 both have narrow critical paths, but they also have many operations that can execute concurrently. Schedules that do

not attempt to conserve registers allow the non-critical-path operations to start long before their results are needed. This ties up registers until the results are used, but it also insures that most of the critical-path operations run without interference from non-critical-path operations.

The parallel kernels require between eight and 256 registers in order to run at their ultimate speeds, depending on the start-limits. With only eight registers, most of the parallel kernels run 2-2.5 times longer than with an unlimited number of registers. Most of the parallel kernels have wide critical paths, and their execution times are often limited by the time required to start all the operations. Thus, many operations are executing at a time, and each operation is allocated a result register. Also, operations do not always start as soon as data dependencies allow because of the limited start-bandwidth, and result registers cannot be freed until the operations that use the results start.

The fact that programs scheduled to use few registers run slower than programs scheduled without regard to register usage provides a motivation for machines with many registers. Also, if the machine has a sufficient number of registers, scheduling time can be spent increasing execution speed, rather than minimizing register usage.

It should be noted that the Cray-1 has a total of 512 vector register elements; therefore much vectorizable code could potentially execute as fast as it could with an unlimited number of registers. However, unvectorized loops like kernel 14 could not make effective use of the vector registers, and therefore, more registers or a more general register structure than the Cray-1 has are needed for the fastest execution of Livermore Kernels 1-14.

When the number of registers is restricted to four, execution times of most kernels are increased by a factor of 1.5 to 3. The execution times of the kernels would be even greater with fewer registers, suggesting that pure memory-to-memory scalar architectures are terribly inefficient, at least for programs with characteristics similar to Livermore Kernels 1-14.

3.4.2. Fast-Register Requirements

During serial sections of programs, when few operations are ready to execute, the operation-execution times dominate the total execution time, so register access should be as fast as possible. During parallel sections of programs, however, longer register access times can be tolerated by overlapping the execution of more operations.

Fast register requirements are tracked by modifying the execution simulator in the following manner. Assume slow registers take P_r more ticks than fast registers to read, and P_w more ticks than fast registers to write. Then, if an operation reads any operands from slow registers, its execution time is increased by P_r ticks. In addition, each time an operation writes a result to a slow register, its execution time is increased by P_w ticks.

Initially, all operations are assumed to use slow registers. Each clock tick, if there are fewer than start-limit critical-path operations in the ready list, the active list is searched for operations that are within P_w ticks of finishing and that allow critical-path operations to be added to the ready list when they finish. One such operation is retroactively allocated a fast register and removed from the active list, and one or more critical-path operations that it sources are added to the ready list. This process is repeated until there are start-limit critical-path operations in the ready list, or until no more almost-finished active operations remain.

When an operation using a fast-register operand starts, it does so without a slow-register-read penalty; the fast register is then released. Thus, fast registers are allocated and reserved from the time they are written until they are first used.

The maximum number of fast registers reserved at a time is tracked for execution with start-limits of one, two, four, and eight, for all 14 kernels. A count of the number of times fast registers are reserved is also maintained, as an indication of the importance of fast registers to fast execution of the program. For example, if fast registers are allocated 2000 times

even though only two fast registers are required, the fast registers speed up at least 2000 critical-path operations, and, thus, have a significant effect on performance. If fast registers are allocated only ten times, they only speed up the computation rarely, and they have a negligible effect on performance.

These data are shown for $P_r = P_w = 1$ in table 3.8. The maximum number of registers used is no more than start-limit, because at most start-limit fast registers are reserved at a time, and all are freed when the start-limit critical operations start the next clock tick.

Table 3.8: Number of Fast Registers Required and Number of Times Fast Registers Used (Reservation from Write to First Result Read, Slow-Register Read and Write Penalties = 1).

Kernel	Number Parallel Starts							
	1		2		4		8	
	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used
1	1	2	2	3	4	5	8	9
2	1	1	2	2	4	4	8	68
3	1	9	1	9	1	9	1	9
4	1	11	1	11	1	11	1	11
5	1	1993	1	1992	1	1992	1	1992
6	1	1995	1	1994	1	1994	1	1994
7	1	2	2	3	4	5	8	9
8	1	2	2	3	4	5	8	9
9	1	1	2	2	4	4	8	8
10	1	8	2	16	4	32	8	64
11	1	1000	1	999	1	999	1	999
12	0	0	0	0	0	0	0	0
13	1	364	2	391	2	391	2	391
14	1	606	2	606	2	607	2	607

Kernels 5, 6, 11, 13, and 14, with very narrow critical paths, can make use of no more than one or two fast registers, although they use them for the result of almost every critical-path operation. One or two fast registers speed up the execution of these kernels significantly.

Kernels 3 and 4 rarely use fast registers, and kernel 12 never uses a fast register. Kernels 3 and 4 essentially execute summation trees, and virtually all operations are in the critical path, so there can be a shortage of critical-path operations only at the bottom of their sum-

mation trees. Every operation in kernel 12 is in the critical path, which is many times wider than the largest start-limit of eight. Kernels 3, 4, and 12 do not need fast registers for fast execution.

Traditionally, a register is reserved for an operation from the time the operation starts until its result is used, rather than from the time a result is written until it is used. This traditional reservation scheme requires that a destination be available before each operation starts, while the time-of-write reservation scheme requires that a destination be available before the operation finishes. Time-of-start reservation thus forces each destination register to sit unused while its operation executes.

If the execution timing of the operations of interest can be predicted at compile time, operation scheduling and reservation checking can be performed at compile time, and there is no disadvantage to time-of-write reservation.

If timing is not predictable at compile time, some run-time checking is necessary for operations to start sooner than worst-case timing requires. Time-of-write reservation requires checking and updating reservations twice for each operation (before it starts and before it finishes), while time-of-start reservation requires checking and updating reservations only once (before the start of each operation). Thus, with overlapped execution of operations and unpredictable execution times, time-of-write reservation requires checking and updating reservation status twice each clock tick, a significant disadvantage over time-of-start reservation.

The next analyses are the same as the previous analyses, except that registers are reserved from the time the operation starts until the result is used. Results are summarized in table 3.9.

These results are essentially the same as for time-of-write register reservation, except that kernel 2 can use as many as 40 or more fast registers if n and start-limit are high enough. The reason for this behavior is that kernel 2 computes the inner products of sub-

vectors of length five, and the unbalanced summation tree causes some results to wait longer before they are used. Better scheduling of this kernel would eliminate the anomalous behavior.

Table 3.9: Number of Fast Registers Required and Number of Times Fast Registers Used (Reservation from Start to First Result Read, Slow-Register Read and Write Penalties = 1).

Kernel	Number Parallel Starts							
	1		2		4		8	
	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used
1	1	2	2	3	4	5	8	9
2	1	1	2	2	4	4	40	68
3	3	9	3	9	3	9	3	9
4	4	11	3	11	3	11	3	11
5	1	1993	1	1992	1	1992	1	1992
6	1	1995	1	1994	1	1994	1	1994
7	1	2	2	3	4	5	8	9
8	1	2	2	3	4	5	8	9
9	1	1	2	2	4	4	8	8
10	1	8	2	16	4	32	8	64
11	1	1000	1	999	1	999	1	999
12	0	0	0	0	0	0	0	0
13	2	364	2	391	2	391	2	391
14	2	606	2	606	2	607	2	607

These results summarized in tables 3.8 and 3.9 show that slower register-access times can be tolerated if fast registers are available for use in the narrower critical paths of the program. Traditional time-of-start register reservation can be used without requiring more registers than would be needed for the more difficult time-of-write register reservation.

Execution times of all kernels with mostly slow registers are shown in table 3.10, along with the times for execution with all fast registers. The number of fast registers are the same as they are in table 3.9, and the slow-register penalty is one clock tick for read and one clock tick for write. Almost without exception, the programs run no slower with mostly slow registers than with all fast registers. In the worst case, kernel 3 with a start-limit of eight requires 3% more time to execute with mostly slow registers than with all fast registers.

Table 3.10: Times for Simulated Execution with Unlimited Fast Registers and with Few Fast Registers and Unlimited Slow Registers (Slow-Register Read and Write Penalties = 1).

Kernel	Number Parallel Starts											
	1			2			4			8		
	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio
1	3216	3217	1.00	1614	1615	1.00	813	814	1.00	413	414	1.00
2	802	803	1.00	402	403	1.00	202	203	1.00	106	108	1.02
3	4120	4121	1.00	2078	2081	1.00	1060	1065	1.00	554	561	1.01
4	1554	1553	1.00	788	791	1.00	410	415	1.01	224	231	1.03
5	12963	12963	1.00	12962	12964	1.00	12962	12964	1.00	12962	12964	1.00
6	12976	12976	1.00	12975	12977	1.00	12975	12977	1.00	12975	12977	1.00
7	2410	2411	1.00	1206	1207	1.00	604	605	1.00	303	304	1.00
8	2064	2065	1.00	1033	1034	1.00	518	519	1.00	260	261	1.00
9	2810	2811	1.00	1406	1407	1.00	704	705	1.00	353	354	1.00
10	2902	2903	1.00	1452	1453	1.00	727	728	1.00	365	366	1.00
11	6009	6009	1.00	6008	6010	1.00	6008	6010	1.00	6008	6010	1.00
12	3003	3004	1.00	1503	1504	1.00	753	754	1.00	378	379	1.00
13	3458	3459	1.00	2484	2488	1.00	2484	2488	1.00	2484	2488	1.00
14	3794	3796	1.00	3793	3793	1.00	3793	3793	1.00	3793	3793	1.00

The simulated execution times using an unlimited number of registers are within 3% of the lower bounds on execution time predicted by theorem 3.7, for all kernels except kernels 13 and 14, as summarized in Appendix C. Furthermore, the simulated execution time for kernel 13 with unlimited registers and a start-limit of one achieves the lower bound, and the simulated execution time for kernel 14 with unlimited registers and a start-limit of one is within 5% of the lower bound. Therefore, these results are not simply artifacts of the simulator.

The results of simulations when the read penalty P_r is increased to two are summarized in tables 3.11 and 3.12. The numbers of fast registers required are essentially the same as for the case when $P_r = 1$, except that kernel 2 uses many fast registers when start-limit is four, as well as when it is eight. The execution times are somewhat higher for $P_r = 2$, representing a worst case degradation of 5% for kernel 4 and a worst case degradation of 12% for kernel 14. For most of the kernels most of the time, however, the execution times and fast register usages are the same as for $P_r = 1$.

Table 3.11: Number of Fast Registers Required and Number of Times Fast Registers Used (Reservation from Start to First Result Read, Slow Register Read Penalty = 2, Slow Register Write Penalty = 1).

Kernel	Number Parallel Starts							
	1		2		4		8	
	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used
1	1	2	2	3	4	5	8	9
2	1	1	2	2	4	4	40	68
3	4	10	4	10	4	10	4	10
4	4	10	4	12	4	12	4	12
5	1	1993	1	1992	1	1992	1	1992
6	1	1995	1	1994	1	1994	1	1994
7	1	2	2	3	4	5	8	9
8	1	2	2	3	4	5	8	9
9	1	1	2	2	4	4	8	8
10	1	8	2	16	4	32	8	64
11	1	1000	1	999	1	999	1	999
12	0	0	0	0	0	0	0	0
13	2	393	2	391	2	391	2	391
14	2	606	2	458	2	459	2	459

Table 3.12: Times for Simulated Execution with Unlimited Fast Registers and with Few Fast Registers and Unlimited Slow Registers, Slow Register Read Penalty = 2, Slow Register Write Penalty = 1.

Kernel	Number Parallel Starts											
	1			2			4			8		
	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio
1	3216	3218	1.00	1614	1616	1.00	813	815	1.00	413	415	1.00
2	802	804	1.00	402	404	1.00	202	204	1.01	106	109	1.03
3	4120	4121	1.00	2078	2082	1.00	1060	1067	1.01	554	564	1.02
4	1554	1555	1.00	788	793	1.01	410	418	1.02	224	235	1.05
5	12963	12963	1.00	12962	12965	1.00	12962	12965	1.00	12962	12965	1.00
6	12976	12976	1.00	12975	12978	1.00	12975	12978	1.00	12975	12978	1.00
7	2410	2412	1.00	1206	1208	1.00	604	606	1.00	303	305	1.01
8	2064	2066	1.00	1033	1035	1.00	518	520	1.00	260	262	1.01
9	2810	2812	1.00	1406	1408	1.00	704	706	1.00	353	355	1.01
10	2902	2904	1.00	1452	1454	1.00	727	729	1.00	365	367	1.01
11	6009	6009	1.00	6008	6011	1.00	6008	6011	1.00	6008	6011	1.00
12	3003	3005	1.00	1503	1505	1.00	753	755	1.00	378	380	1.01
13	3458	3460	1.00	2484	2490	1.00	2484	2490	1.00	2484	2490	1.00
14	3794	3950	1.04	3793	4238	1.12	3793	4238	1.12	3793	4238	1.12

Table 3.13: Number of Fast Registers Required and Number of Times Fast Registers Used (Reservation from Start to First Result Read, Slow Register Read Penalty = 4, Slow Register Write Penalty = 1).

Kernel	Number Parallel Starts							
	1		2		4		8	
	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used	Max Req.	Times Used
1	1	2	2	3	4	5	8	9
2	1	1	2	2	28	40	40	68
3	5	12	5	12	5	12	5	12
4	5	14	5	14	5	14	5	14
5	1	1993	1	1992	1	1992	1	1992
6	1	1995	1	1994	1	1994	1	1994
7	1	2	2	3	4	5	8	9
8	1	2	2	3	4	5	8	9
9	1	1	2	2	4	4	8	8
10	1	8	2	16	4	32	8	64
11	1	1000	1	999	1	999	1	999
12	0	0	0	0	0	0	0	0
13	2	382	2	391	2	391	2	391
14	2	606	2	606	2	607	2	607

Table 3.14: Times for Simulated Execution with Unlimited Fast Registers and with Few Fast Registers and Unlimited Slow Registers, Slow Register Read Penalty = 4, Slow Register Write Penalty = 1.

Kernel	Number Parallel Starts											
	1			2			4			8		
	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio	All Fast	Few Fast	Ratio
1	3216	3220	1.00	1614	1618	1.00	813	817	1.00	413	417	1.01
2	802	806	1.00	402	406	1.01	202	204	1.01	106	112	1.06
3	4120	4122	1.00	2078	2085	1.00	1060	1072	1.01	554	571	1.03
4	1554	1554	1.00	788	797	1.01	410	424	1.03	224	243	1.08
5	12963	12963	1.00	12962	12967	1.00	12962	12967	1.00	12962	12967	1.00
6	12976	12976	1.00	12975	12980	1.00	12975	12980	1.00	12975	12980	1.00
7	2410	2414	1.00	1208	1210	1.00	604	608	1.01	303	307	1.01
8	2064	2068	1.00	1033	1037	1.00	518	522	1.01	260	264	1.02
9	2810	2814	1.00	1406	1410	1.00	704	708	1.01	353	357	1.01
10	2902	2906	1.00	1452	1456	1.00	727	731	1.01	365	369	1.01
11	6009	6009	1.00	6008	6013	1.00	6008	6013	1.00	6008	6013	1.00
12	3003	3007	1.00	1503	1507	1.00	753	757	1.01	378	382	1.01
13	3458	3462	1.00	2484	2494	1.00	2484	2494	1.00	2484	2494	1.00
14	3794	4551	1.20	3793	4398	1.16	3793	4395	1.16	3793	4394	1.16

Results when the read penalty P_r is increased to four are summarized in tables 3.13 and 3.14. For $P_r = 4$ the execution times are as much as 20% greater than execution times when

$P_r = 1$, so slow registers with access times that are this slow are not nearly as useful as faster registers.

These results suggest that three different register speeds could be useful: a small set of sub-clock-tick registers for the most critical operations, a larger set of 1-tick registers for moderately critical operations, and an even larger set of 2-tick registers for the non-critical operations.

The results summarized in figures 3.2a-3.2n and in tables 3.9-3.14 suggest that a small set of high speed registers supplementing a large set of slower registers provide the same performance as an unlimited set of high speed registers. This performance is about a factor of three faster than if only eight registers are provided.

3.5. Summary of Program Characteristics and Hardware Requirements

The important program characteristics of the Livermore Kernels⁶⁷ are summarized below. The diversity of program characteristics suggests that this set of program kernels represents a significant fraction of scientific programs.

- (1) Serial programs need short operation-execution times for fast execution, while parallel programs need to start many operations at a time for fast execution.
- (2) Many memory accesses can be started well before their results are needed, so higher-latency, high-bandwidth memory can be used for them.
- (3) Several hundred temporary results could be maintained by some programs with some schedules, so several hundred temporary storage locations that are faster than main memory could be effectively used to speed up execution.
- (4) Most results are used immediately after they are generated, so much of the temporary storage could be optimized for speed and density, without regard for long-term data integrity.

- (5) Most results are only used once, so specialized or implicit result-transmission could be used much of the time.
- (6) Hardware requirements and execution times of programs can be accurately predicted from the partitionings of the dependency graphs. Analyses of these condensed specifications of programs are both faster and more generally applicable than analyses and simulations of the full dependency graphs.
- (7) Programs executing with mostly slow registers and a few fast registers can run essentially as fast as programs executing with all fast registers, for programs that can be vectorized and for those that cannot be vectorized.
- (8) More registers than the Cray-1 provides or a more general access mechanism are needed for maximum performance of scientific programs, especially for those programs that cannot be vectorized.
- (9) Computers with pure memory-to-memory scalar architectures have a maximum execution speed that is at most one third to one half the speed of computers with many programmer-addressable registers.

CHAPTER 4

Datapath Design

The time to execute an operation includes the time to read the operands from registers, propagate the data to the functional units, generate a result, propagate the result to the destination, and write the result. In an efficient datapath design, signal-propagation delays and device-selection delays are minimized, and functional unit latency/throughput tradeoffs are optimized for the functions and expected workload. Signal-propagation delays include electrical-propagation delays which are largely determined by the propagation media and device placement, and gate delays. Device-selection delays are determined by the gate delays and fan-in and fan-out limitations of gates. Minimizing pipeline latency and maximizing pipeline throughput require trading off the number of pipeline stages with the pipeline latch overhead, and optimal choices depend on both the functions implemented and the workload characteristics.

The results developed in this chapter are used in Chapter 5, where temporary storage structures are developed, and in Chapter 7, where an architecture is proposed.

4.1. Signal-Propagation Delays

Signals must propagate through gates and wires before they reach their destinations. Gate delays are determined by the gate technology and by the loading of the output. All delays are measured in terms of the basic gate delay for the technology used, and all wires are assumed to behave as properly terminated transmission lines, so end effects such as those caused by gate input capacitance are not considered; the effects of capacitive loading can be included by increasing the basic gate delay to that of a gate with an "average" output loading. Signal-propagation delays in wires are determined by the speed of the propagation

media, the placement of devices, and the wiring technology. Propagation speed is determined by the propagation medium, and is limited by the speed of light in a vacuum; speeds for several popular interconnection media are shown in table 4.1.

Table 4.1: Propagation Speeds and Delays for Various Media.

Medium	Speed	Delay
light in vacuum	3.0×10^8 m/s	33 ps/cm
coax (foam)	2.5×10^8 m/s	40 ps/cm
microstrip (teflon)	2.4×10^8 m/s	43 ps/cm
stripline (teflon)	1.8×10^8 m/s	51 ps/cm
microstrip (glass-epoxy)	1.7×10^8 m/s	59 ps/cm
stripline (glass-epoxy)	1.3×10^8 m/s	71 ps/cm.
chip interconnect	4.2×10^7 m/s	240 ps/cm

For propagation between random pairs of devices, the propagation distances grow as a function of the number of devices. In a plane, distances grow at a rate between the square root and linearly with the number of devices. In three dimensions, distances grow at a rate between the cube root and the square root of the number of devices.

Consider the plane first, which includes both printed-circuit and integrated-circuit wiring. Suppose there are k devices, each with area a , and suppose that each device is connected to c other devices with wires of width w and average length l , using p wiring planes that are independent of the devices and of each other. The area of devices is ka , while the area of wiring is $\frac{cklw}{p}$. Assuming the average wire length is r , the radius of the circle containing k devices is

$$r = \left[\frac{ka}{\pi} \right]^{1/2} \quad (4.1)$$

if wire area is negligible compared to device area. Assuming the average wire length is equal to the radius, the propagation distance grows at a rate no slower than the square root of the number of devices. If wire area is not negligible, the radius of the circle containing the wiring for k devices is

$$r = \left[\frac{ckrw}{p\pi} \right]^{1/2} = \frac{ckw}{p\pi}, \quad (4.2)$$

so the propagation distance grows at a rate at least linear with the number of devices.

Now consider a three-dimensional enclosure. Suppose there are k devices, each with volume v , and suppose that each device is connected to c other devices with wires of cross-sectional area a and average length r . The radius of the sphere that contains k devices is

$$r = \left[\frac{3kv}{4\pi} \right]^{1/3}, \quad (4.3)$$

so, if wire volume is negligible, the average propagation distance grows at a rate no slower than the cube root of the number of devices. The radius of the sphere containing just the wiring is

$$r = \left[\frac{3ckra}{4\pi} \right]^{1/3} = \left[\frac{3cka}{4\pi} \right]^{1/2}, \quad (4.4)$$

so, if wire volume is not negligible, the average propagation distance grows at a rate at least as fast as the square root of the number of devices.

As the number of registers, latches, functional units, etc. increase, the longer propagation delays increase operation execution times, which increase the total execution times of serial sections of programs, as was shown in Chapter 3.

The execution time of every operation is the sum of register access times and computation times; since every operation accesses a register at least once (and many operations access registers both at the beginning and at the end of their execution), its execution time is at least

$$etime_j = t_{acc} + t_{comp}. \quad (4.5)$$

The register access time is the sum of register selection time and propagation time between register and functional unit

$$t_{acc} = t_{select} + t_{prop}. \quad (4.6)$$

Suppose that the number of registers is increased by a factor of c . Assume that the selection time t_{select} does not increase as the number of registers increases, and that the propagation time increases at the most optimistic rate of $c^{1/3}$; the new register access time

$$t_{\text{acc}_{\text{new}}} = t_{\text{select}} + c^{1/3}t_{\text{prop}}, \quad (4.7)$$

so all execution times are increased by at least $(c^{1/3}-1)t_{\text{prop}}$:

$$etime_{j_{\text{new}}} = etime_j + (c^{1/3}-1)t_{\text{prop}}. \quad (4.8)$$

The upper bound on computation time given by theorem 3.6 is increased to

$$\left[\sum_{i=1}^p \left(\frac{opcount_i}{start-limit} + etime_{latest_i} + (c^{1/3}-1)t_{\text{prop}} \right) \right] \text{ ticks}, \quad (4.9)$$

an increase of $p(c^{1/3}-1)t_{\text{prop}}$ ticks. The lower bounds on computation time given by theorem 3.7 is similarly increased by $p(c^{1/3}-1)t_{\text{prop}}$ ticks.

Suppose t_{prop} is .25 ticks (on the Cray X-MP this corresponds to 2125 picoseconds, or 36 centimeters propagation distance) and that c is 256, increasing the register count from eight to 2048. This increases the upper and lower bounds on computation by $1.3p$ ticks. Livermore Kernel 12, with three partitions, would run at least four ticks longer with the extra registers. Livermore Kernel 5, with 2666 partitions, would run at least 1994 ticks longer, a 21% increase, and Livermore Kernel 11, with 1001 partitions, would run at least 1338 ticks longer, a 22% increase. Thus, parallel programs are hardly affected by the longer propagation times due to more registers, while serial programs are considerably slowed by the longer propagation times.

The propagation times discussed above assume random placement of devices, which is reasonable if all devices are treated uniformly. However, it is possible to organize devices non-uniformly, so that many devices can be accessed without increasing access times of all of them.

This can be accomplished by grouping together those devices among which communication is very important, while allowing other devices to be further apart. Since the volume of a sphere grows as the cube of the radius, the available space for devices increases rapidly as they are moved away from the center of communication. This space allocation tends to create one or more physical clusters of devices, with much communication within the centers of the clusters and less communication among clusters.

Most wiring technologies do not allow devices to be connected via the shortest possible paths. For example, in printed-circuit technology, microstrip lines must be routed around pins and other lines, so lines are usually longer than the Euclidean distances between pairs of devices. Most wiring between boards is via connectors at the board edges, rather than directly through the boards. Cabling may be longer than their minimum lengths, and may be routed along the periphery of the computer to facilitate servicing. In a successful high-speed design, extra wiring delays are kept small, but they are always present.

4.2. Data Routing

4.2.1. Basic Selection Delays

Data routed among sources and destinations are subject to logic delays in addition to signal-propagation delays. In his text on computer architecture, Kuck⁵⁵ shows that the longest path of a k -way fan-in or fan-out of signals, using components with a fan-in or fan-out of f , requires $\lceil \log_f(k) \rceil$ component delays.

For example, sending a datum from one source to up to eight destinations using gates with maximum fan-out=2 requires three levels of logic, as shown in figure 4.1.

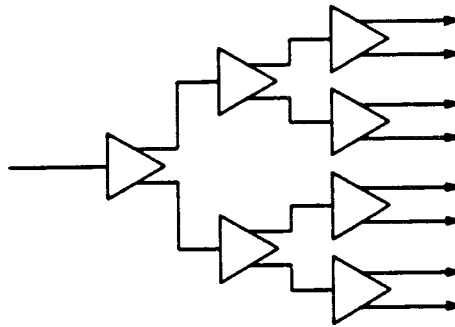


Figure 4.1: Fanning a Datum Out from One Source to Eight Destinations, Using Gates with a Maximum Fan-Out of 2.

For another example, multiplexing eight data down to one using gates with maximum fan-in=3 requires one gate delay to AND the data with a select signal, plus two levels of OR logic to fan the eight signals down to one, as shown in figure 4.2.

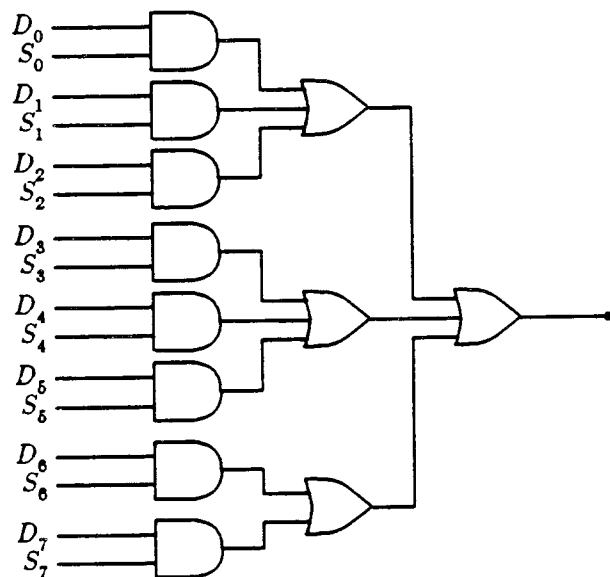


Figure 4.2: Multiplexing Eight Data Down to One, Using Gates with a Maximum Fan-In of 3.

Without a pure OR or pure AND function, a level of inverters would be necessary after each NOR or NAND gate in the OR fan-in tree, increasing the total number of gate delays for the multiplexer to $\lceil 2 \log_{\text{fan-in}(k)} \rceil$. If the eight-to-one multiplexer above is constructed

using only NAND or NOR gates with fan-in=3, it would require four gate delays.

The delay in computing the data select signals S_i is not included in the multiplexing time because multiplexer addresses are often data-independent, so they can be computed at compile time and specified sufficiently far ahead in the instruction stream that their decoding does not slow the computation. Those cases when device addresses are data dependent and decoding times are important are considered below.

A j -bit address determines which data input is selected by a $2^j:1$ multiplexer. The initial selection gate of the multiplexer can AND each data input with up to fan-in-1 select inputs, so the j bits of address can be decoded as fan-in-1 separate $d:2^d$ partial decoders, where

$$d = \left\lceil \frac{j}{(\text{fan-in}-1)} \right\rceil. \text{ Each of these partial decoder outputs is used by } 2^{\text{fan-in}-2} \text{ selection gates,}$$

requiring an additional fan-out tree of height

$$\left\lceil \log_{\text{fan-out}} \left\{ 2^{\text{fan-in}-2} \right\} \right\rceil - 1 = \left\lceil \frac{\text{fan-in}-2}{\log_2(\text{fan-out})} \right\rceil - 1. \quad (4.10)$$

Within each partial decoder, the $\left\lceil \frac{j}{(\text{fan-in}-1)} \right\rceil$ inputs are decoded using AND trees of

height $\left\lceil \log_{\text{fan-in}} \left\lceil \frac{j}{(\text{fan-in}-1)} \right\rceil \right\rceil$. Each input or its complement is used by $2^{\text{fan-in}-1}$ AND gates,

so fan-out trees of height $\left\lceil \frac{\text{fan-in}-1}{\log_2(\text{fan-out})} \right\rceil$ precede the fan-in tree. Thus, the decoder delay

before the selection gate of a multiplexer is

$$\left\lceil \frac{\text{fan-in}-1}{\log_2(\text{fan-out})} \right\rceil + \left\lceil \log_{\text{fan-in}} \left\lceil \frac{j}{(\text{fan-in}-1)} \right\rceil \right\rceil + \left\lceil \frac{\text{fan-in}-2}{\log_2(\text{fan-out})} \right\rceil - 1 \quad (4.11)$$

gate delays. This analysis assumes that pure OR gates and pure AND gates are available.

A 4:1 multiplexer including the decoder circuitry, using gates with maximum fan-out=2 and fan-in=3, is shown in figure 4.3.

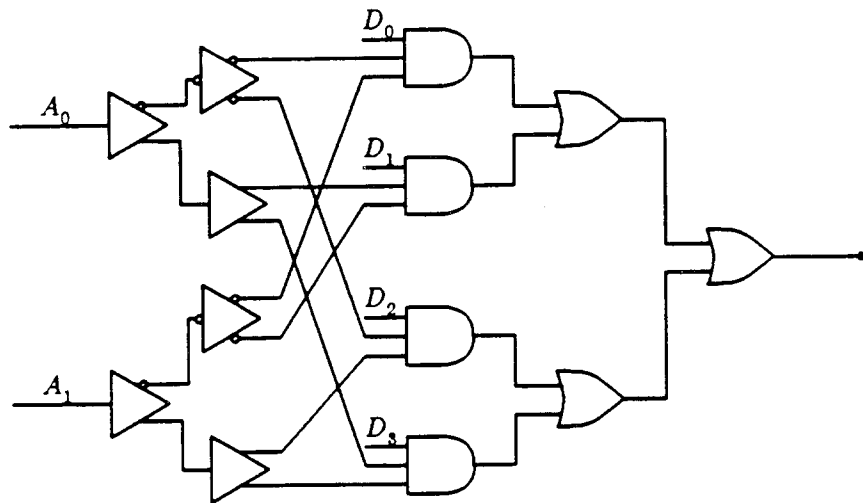


Figure 4.3: 4:1 Multiplexer with Decoding Circuitry, Using Gates with a Maximum Fan-Out of 2 and a Maximum Fan-In of 3.

A_0 and A_1 are each fanned out to four signals using two levels of gates. The fan-in is equal to $j+1$, so no partial decoding is necessary before the select gates. Finally, the selected signals are ORed using an OR fan-in tree of height two.

4.2.2. Unbalanced Selection Trees

The fan-in trees and fan-out trees described in the previous section all are as balanced as possible, and this minimizes the maximum tree heights. It is possible to construct unbalanced fan-in and fan-out trees that allow shorter paths for some signals.

With unbalanced fan-out trees, the first fan-out ^{i} - a signals are generated after i gate delays, followed by a fan-out ^{j} - b signals generated after $i + j$ gate delays, followed by b fan-out ^{k} - c signals generated after $i + j + k$ gate delays, and so on. An unbalanced fan-out tree is shown in figure 4.4 for fan-out = 2, $i = 2$, $a = 2$, $j = 2$, and $b = 0$. Thus two signals are available after two gate delays, and eight signals are available after four gate delays.

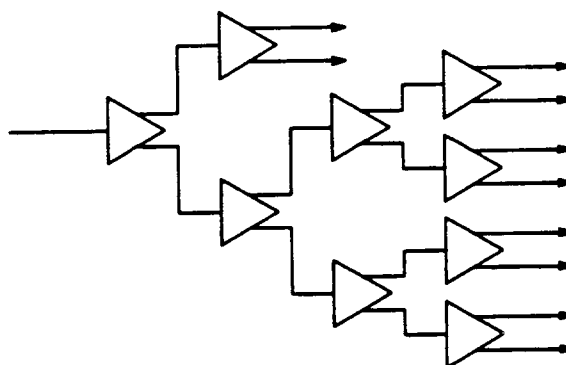


Figure 4.4: Unbalanced Fan-Out Tree, Fanning a Datum Out from One Source to Ten Destinations, Using Gates with a Maximum Fan-Out of 2.

With unbalanced fan-in trees, fan-in^{*i*} - *a* signals can be selected after *i* gate delays, followed by *a* fan-in^{*j*} - *b* signals selected after *i* + *j* gate delays, followed by *b* fan-in^{*k*} - *c* signals selected after *i* + *j* + *k* gate delays, and so on. An unbalanced fan-in tree is shown in figure 4.5 for fan-in = 2, *i* = 2, *a* = 2, *j* = 2, and *b* = 2.

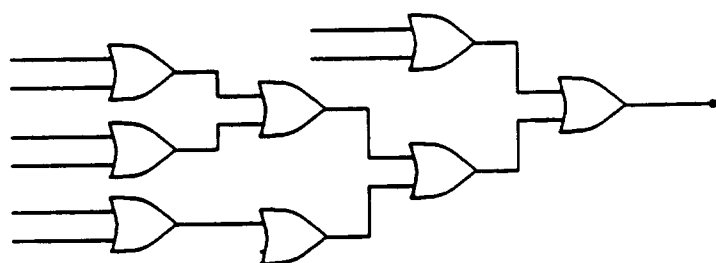


Figure 4.5: Unbalanced Fan-In Tree, Fanning Data from Eight Sources In to One, Using Gates with a Maximum Fan-In of 2.

Here two signals are available after two gate delays, and six signals are available after four gate delays.

It is difficult to design a decoder that has a shorter delay for some outputs because all address bits are usually equally important, so the AND fan-in trees must all have equal breadths and, hence, depths. However, addresses are often data-independent, so decoding times are not very important, as discussed in the previous section. Also, the delay through

AND fan-in trees is $\left\lceil \log_{\text{fan-in}} \left\lceil \frac{\log_2(k)}{(\text{fan-in}-1)} \right\rceil \right\rceil$, which is much smaller than the $\left\lceil \log_{\text{fan-in}}(k) \right\rceil$ delay

through the OR fan-in trees for multiplexers if k is large enough, so reducing decoding times is less important than reducing multiplexing times.

For those cases where addresses are data-dependent and where decoding times must be reduced for some devices, redundant address representations, similar to Huffman encoding,⁴⁸ can be used. For example, a four-bit address can be used to specify one of eight devices, using the mapping shown in table 4.2:

Table 4.2: Mapping of Address Bits to Devices.

a_3	a_2	a_1	a_0	device
0	0	x	x	0
0	1	x	x	1
1	0	0	0	2
1	0	0	1	3
1	0	1	0	4
1	0	1	1	5
1	1	0	0	6
1	1	0	1	7

The decoding circuitry for devices 0 and 1 use only the two address bits a_0 and a_1 , whereas the decoding circuitry for devices 2 through 7 use all four address bits.

Two disadvantages of redundant addresses are that their maximum lengths are longer than non-redundant addresses and that they have variable lengths. Variable-length addresses require that instructions have variable lengths and variable formats, both of which complicate the instruction fetching and decoding process. If all addresses have the maximum length, with some bits of some addresses ignored, an instruction-stream bandwidth that is higher than is necessary for non-redundant addresses is required. Because decoding time is relatively insignificant, as discussed above, redundant addresses are not considered further.

4.2.3. Data-Routing Delays

For the following examples, AND, OR, NAND, and NOR gates are assumed. They have maximum fan-in = 5, and maximum fan-out = 32. These characteristics are typical of modern

ECL circuits.^{35,70,89}

Consider selection from among eight devices, using three-bit addresses. A 1:8 fan-out tree has a delay of $\left\lceil \log_{32}(8) \right\rceil = 1$ gate delays. An 8:1 multiplexer has a delay of $\left\lceil \log_8(8) \right\rceil + 1 = 3$ gate delays. A 3:8 decoder has a delay of

$$\left\lceil \frac{4}{\log_2(32)} \right\rceil + \left\lceil \log_8 \left\lceil \frac{3}{4} \right\rceil \right\rceil + \left\lceil \frac{3}{\log_2(32)} \right\rceil = 2$$

gate delays.

Now consider selection from among 2048 devices, using 11-bit addresses and only balanced fan-in and fan-out trees. A 1:2048 fan-out tree has a delay of $\left\lceil \log_{32}(2048) \right\rceil = 3$ gate delays. A 2048:1 multiplexer has a delay of $\left\lceil \log_8(2048) \right\rceil + 1 = 6$ gate delays. An 11:2048 decoder has a delay of

$$\left\lceil \frac{4}{\log_2(32)} \right\rceil + \left\lceil \log_8 \left\lceil \frac{11}{4} \right\rceil \right\rceil + \left\lceil \frac{3}{\log_2(32)} \right\rceil = 3$$

gate delays. Note that the fan-out and fan-in delays are more than double those for the eight-device case, while the decoding delay is only one more than for the eight-device case.

Now consider fan-out and fan-in trees that are unbalanced, so that eight devices can be accessed quickly and 2040 devices are accessed less quickly. A signal can be fanned out to the eight fast-access devices in one gate delay, so the signal can be fanned out to the remaining 2040 devices in

$$1 + \left\lceil \log_{32} \left\lceil \frac{2040}{8} \right\rceil \right\rceil = 3$$

gate delays. One of the eight fast-access devices can be selected in three gate delays, while one of the remaining 2040 devices can be selected in

$$3 + \left\lceil \log_6 \left(\frac{2040}{8} \right) \right\rceil = 7$$

gate delays.

The delays for fanning out, multiplexing, and decoding are summarized in table 4.3.

Table 4.3: Delays for Fanning Out, Multiplexing, and Decoding.

Function	Number of Devices			
	8	balanced	unbalanced	
		2048	first 8	last 2040
fan-out	1	3	1	3
multiplexer	3	6	3	7
decoder	2	3	3	3

Thus, unbalanced fan-in and fan-out trees make it is possible to access a few devices very quickly and still access many more devices through the same data paths, almost as fast as if balanced fan-in and fan-out trees are used.

4.3. Pipeline Design and Analysis

A logical function implemented using L levels of logic can be pipelined into $S \leq L$ stages of logic, with a pipeline latch after each stage. The throughput of the unpipelined logical function is $1/L$ results per gate delay, so a theoretical throughput of S/L results per gate delay could be achieved by pipelining into S stages. In practice, the overhead for a pipeline latch limits S to a fraction of L .

Earle latches and polarity-hold latches have the advantage that the gates in the latches can perform useful logical functions. A simple Earle latch is shown in figure 4.6. When C is high and \bar{C} is low, the D input appears at the output. When C is low and \bar{C} is high, the output feeds back and is latched. The middle gate eliminates the hazard caused by C going low before \bar{C} goes high.

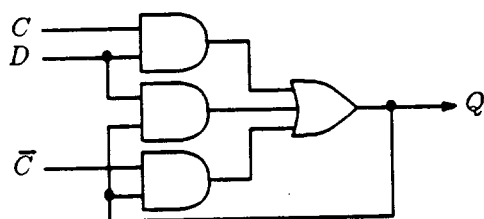


Figure 4.6: Earle Latch Implemented Using AND Gates and OR Gates.

An Earle latch that computes the exclusive-OR of X and Y is shown in figure 4.7. $\bar{X}YC$ and $X\bar{Y}C$ are computed by the top two AND gates, and are summed by the OR gate to form $Q = X\bar{Y} + \bar{X}Y$ when $C = 1$. When $C = 0$, Q feeds back to the output. The logic hazard caused by C going low before \bar{C} goes high is prevented by the two AND gates computing $\bar{X}YQ$ and $X\bar{Y}Q$. If the fan-in of the OR gate is less than five, a fan-in tree must be used instead of the single OR gate.

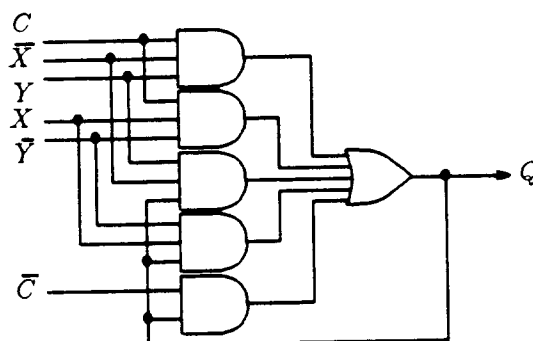


Figure 4.7: Earle Latch that Computes the Exclusive-OR of X and Y .

A polarity-hold latch is shown in figure 4.8. Relative to the Earle latch of figure 4.6, an OR gate input and an AND gate is saved for each data input to each polarity-hold latch, reducing the number of gates and the height of the OR fan-in trees. Proper operation is insured if C always stays high until \bar{C} goes high. Polarity-hold latches are used almost universally in modern supercomputers, so further discussion will concentrate on them.

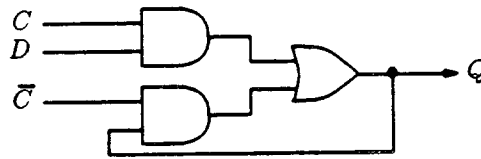


Figure 4.8: Polarity-Hold Latch Implemented Using AND Gates and OR Gates.

4.3.1. Clocking Constraints for Polarity-Hold Latches

Kunkel and Smith⁵⁷ have examined the latch overhead of polarity-hold latches as a function of gate delay uncertainty and clock skew. Their work is briefly summarized in this section.

If unintentional clock skew is zero the clock period must be at least

$$C_{\text{high}} + C_{\text{low}} \geq 2t_{\text{max}} + P_{\text{max}}, \quad (4.12)$$

provided \bar{C} falls no later than $(t_{\text{max}} - t_{\text{min}})$ before C rises, where

C_{high} = duration of $C = 1$.

C_{low} = duration of $C = 0$.

t_{max} = maximum gate delay.

t_{min} = minimum gate delay.

P_{max} = maximum delay between pipeline stages besides the latch delays.

This bound is based on the constraints that C remains high long enough for D to propagate to Q and then to feed back, and that C does not remain high so long that data flows from one latch through the following latch. It further relies on the assumption that $C_{\text{high}} = \bar{C}_{\text{low}}$.

If unintentional clock skew is nonzero, the clock period must be at least

$$C_{\text{high}} + C_{\text{low}} \geq 2t_{\text{max}} + nt_{\text{max}} + U_{C_{i-1}, C_i}, \quad (4.13)$$

provided \bar{C} falls no later than $(t_{\text{max}} - t_{\text{min}}) + U_{C, \bar{C}}$ before C rises, where:

n = number of gate delays between latches.

$r = t_{\text{min}}/t_{\text{max}}$.

U_{C_{i-1}, C_i} = unintentional skew between clocks for stages $i-1$ and i .

$U_{C, \bar{C}}$ = unintentional skew between C and \bar{C} within a latch.

If n is too small, the last constraint must be met by adding delay between pipeline latches.

Thus

$$C_{\text{high}} + C_{\text{low}} \geq (n+2)t_{\text{max}} + U_{C_{i-1}, C_i} \quad (4.14)$$

$$\text{if } n \geq \frac{4}{r} - 4 + \frac{U_{C_{i-1}, C_i} + 2U_{C, \bar{C}}}{rt_{\text{max}}},$$

or

$$C_{\text{high}} + C_{\text{low}} \geq (n+2)t_{\text{max}} + (4 - (n+4)r)t_{\text{max}} + 2U_{C_{i-1}, C_i} + 2U_{C, \bar{C}} \quad (4.15)$$

$$\text{if } n < \frac{4}{r} - 4 + \frac{U_{C_{i-1}, C_i} + 2U_{C, \bar{C}}}{rt_{\text{max}}}.$$

This bound assumes that wire delays (rather than gates) are used to add intentional delays between stages.

If one gate is used to generate \bar{C} , then $U_{C, \bar{C}} = 1/2(1-r)t_{\text{max}}$. If k levels of logic are used to fan out the clock, then $U_{C_{i-1}, C_i} = k(1-r)t_{\text{max}}$. This concludes the summary of Kunkel and Smith's work.

The smallest value of n that does not require extra delay to be added is a function of the gate-delay uncertainty r and the clock uncertainties. If two levels of logic are used to fan out the clocks to each pipeline, then $U_{C_{i-1}, C_i} = 2(1-r)t_{\text{max}}$. Table 4.4 shows the smallest values of n that require no extra delays for various values of r .

Table 4.4: Smallest Numbers of Levels of Logic n that Require No Extra Delays, for Various Gate-Delay Uncertainties r .

r	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
n_{smallest}	63	28	16	11	7	5	3	2	1

Typically r ranges between 0.3 and 0.6 for high-speed integrated circuits.

The wire delays between latches can be used to some advantage besides insuring correct circuit operation. For example, the wire delays can be implemented linearly, so that pipelines are physically stretched out, rather than clustered tightly. Then, the long pipelines can be looped out from a cluster of logic, so only the entrances and exits to the pipelines are adjacent

to the central cluster. Most of the logic for the pipelines would be away from the center where there is more space, as discussed at the end of section 4.1. The pipelines around a central cluster might resemble petals of a flower, as shown in figure 4.9.

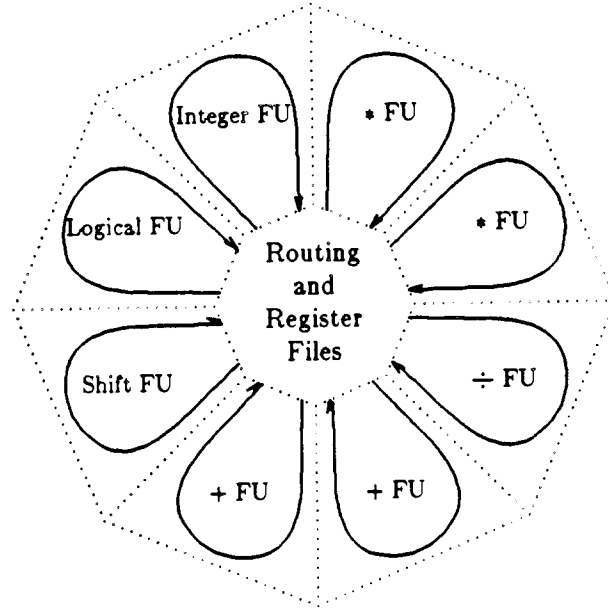


Figure 4.9: Radial Arrangement of Functional Units in a CPU, Minimizing Propagation Distances to and from Registers and Functional Units.

4.3.2. Latency-Throughput Tradeoffs for Pipelines

The latency of a pipeline is the product of the clock period and the number of stages $(C_{\text{high}} + C_{\text{low}})S$, where $S = L/(n + 2)$. The time to produce N independent results is the time to start each operation through the pipeline plus the time to drain the pipeline, for a total time of $T = (N + L/(n + 2))(C_{\text{high}} + C_{\text{low}})$. Substituting $p = n + 2$, T can be expressed as

$$T = (N + L/p)(pt_{\text{max}} + U_{C_{i-1}, C_i}), \quad (4.16)$$

$$\text{if } p \geq \frac{4}{r} - 2 + \frac{U_{C_{i-1}, C_i} + 2U_{C, \bar{C}}}{rt_{\text{max}}},$$

or

$$T = (N + L/p)(pt_{\text{max}} + (4 - (p + 2)r)t_{\text{max}} + 2U_{C_{i-1}, C_i} + 2U_{C, \bar{C}}), \quad (4.17)$$

$$\text{if } p < \frac{4}{r} - 2 + \frac{U_{C_{i-1},C_i} + 2U_{C,\bar{C}}}{rt_{\max}}.$$

The total processing time is minimized when $\frac{dT}{dp} = 0$, or when

$$p = \max \left[2, \left[\frac{LU_{C_{i-1},C_i}}{Nt_{\max}} \right]^{1/2}, \left[\frac{2L((2-r)t_{\max} + U_{C_{i-1},C_i} + U_{C,\bar{C}})}{N(1-r)t_{\max}} \right]^{1/2} \right]. \quad (4.18)$$

The optimal number of stages is

$$S = \left\lceil \frac{L}{p} \right\rceil. \quad (4.19)$$

The optimal number of pipeline stages grows as the square root of the number of levels of gates to implement a logical function, a quantity fixed by the hardware algorithm. The optimal number of pipeline stages also grows as the square root of the number of independent operations, but this quantity is application dependent. For this reason it would be useful to be able to change the length of a pipeline to match the program characteristics: a short pipeline with a slow clock when there is little program parallelism, and a long pipeline with a full-speed clock when there is much program parallelism. This is examined below.

4.3.2.1. Variable-Speed Pipelines

Consider a variable-speed pipeline that runs at either full speed or half speed. During full-speed operation, all latches are controlled by the same system clock. During half-speed operation, some of the latches remain open (C is high, \bar{C} is low), and the remaining latches are controlled by a clock running at one half the system clock rate. During half-speed operation, the pipeline has less than one half its full-speed latency because the latch overhead is less.

Suppose $L = 50$ and $r = 0.3$, and that $U_{C_{i-1},C_i} = 2(1-0.3)t_{\max} = 1.4t_{\max}$, and $U_{C,\bar{C}} = 1/2(1-0.3)t_{\max} = .35t_{\max}$. These values would be typical of the floating-point pipelines of the Cray-1,²¹ if they were implemented with 100K ECL integrated circuits.^{35,89}

Optimal values of p , S , and the corresponding clock periods C and latencies l for various values of N are shown in table 4.5.

Table 4.5: Optimal Numbers of Levels of Logic p per Stage and Numbers of Stages S for Various Numbers of Independent Operations N , with Corresponding Clock Periods C and Latencies L when the Logical Function Requires 50 Levels of Logic and the Gate Speed Uncertainty is 0.3.

	N									
	1	2	3	5	6	14	20	31	55	123
p	25	17	13	10	9	6	5	4	3	2
S	2	3	4	5	6	9	10	13	17	25
C	24.4	18.8	16.0	13.9	13.2	11.1	10.4	9.7	9.0	8.3
l	49	56	64	62	79	100	104	126	153	207

The overhead terms are larger for pipeline stages with fewer gates than for pipeline stages with more gates, and there are more of them; pipelines with many stages have more latency than pipelines with fewer stages. The overhead terms are due to wire delays before the pipeline latches, which prevent data from flowing from one latch through the following latch while C is still high.

This delay in the form of wire pads can be switched in and out using the circuit shown in figure 4.10. An extra gate has been added to the polarity-hold latch, and an extra input has been added to the data AND gates. During full-speed operation $P = 1$ and $\bar{P} = 0$, so data must pass through the wire pad before it enters the OR gate. During half-speed operation $P = 0$ and $\bar{P} = 1$, so data can bypass the wire pad and enter the OR gate directly. (The P input to the top AND gate may not be necessary if care is taken in the design of the control so that a one-glitch cannot emerge from a wire pad after switching from full-speed to half-speed operation.)

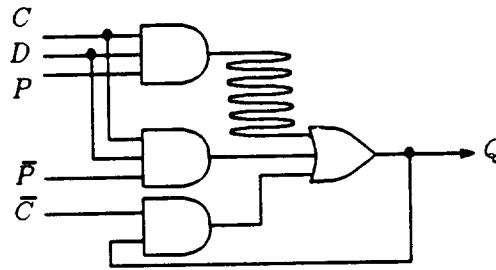


Figure 4.10: Polarity-Hold Latch with Wire-Pad Switching-Circuit.

Theorem 4.1: Adding a wire pad switch circuit to each polarity-hold latch of a pipeline implementing a logical function of L levels of logic increases the depth of logic in the pipeline to at most

$$\frac{2L \log(\text{fan-in}-1)}{\log((\text{fan-in}-2)(\text{fan-in}-1)/2)}$$

levels of logic.

Proof: Assume the logical function is fully pipelined into $L/2$ stages, with each polarity-hold latch implemented as a set of AND gates feeding a single OR gate. One input to each of the AND gates is used for the C or \bar{C} input, so $\text{fan-in}-1$ inputs are left for data. One input to the OR gate is used for the latch output feedback AND gate, so $\text{fan-in}-1$ inputs are left for data. Thus, each polarity-hold latch combines up to $(\text{fan-in}-1)^2$ data inputs to produce one output, so $L/2$ polarity-hold latches can combine up to $(\text{fan-in}-1)^L$ data inputs to produce one bit of output.

When a wire pad switch circuit is added to a polarity-hold latch, another input to each data AND gate is required for the P or \bar{P} input, so $\text{fan-in}-2$ inputs are available for data. For each original padded AND gate an unpadded AND gate must be added, which requires half of the $\text{fan-in}-1$ inputs to the OR gate. Thus, each polarity-hold latch with a wire pad switch circuit can combine up to $(\text{fan-in}-2)(\text{fan-in}-1)/2$ data inputs.

Combining the original $(\text{fan-in}-1)^L$ data inputs then requires no more than

$$\log_{((\text{fan-in}-2)(\text{fan-in}-1)/2)}((\text{fan-in}-1)^L) = \frac{L \log(\text{fan-in}-1)}{\log((\text{fan-in}-2)(\text{fan-in}-1)/2)} \quad (4.20)$$

stages, or

$$\frac{2L \log(\text{fan-in}-1)}{\log((\text{fan-in}-2)(\text{fan-in}-1)/2)} \quad (4.21)$$

levels of logic. If the logical function represents the combination of fewer than $(\text{fan-in})^L$ inputs, or if the logical function is not fully pipelined, then fewer levels of logic are required. ■

Using theorem 4.1 and assuming that gates have an ECL fan-in of five, the length of a pipeline would increase to as much as 1.55 times the original length, if wire pad switch circuits are added. This would increase the latency of the pipeline by as much as 55%, which would nullify the potential benefits of a variable-speed pipeline. When the extra gate cost for the wire pad switch circuit is considered, variable-speed pipelines seem even less attractive. They are not be considered any further.

Corollary 4.2: If a pipeline implementing a logical function of L levels of logic using polarity-hold latches is re-implemented using Earle latches, the depth of logic increases to no more than

$$\frac{2L \log(\text{fan-in}-1)}{\log((\text{fan-in}-1)^2/2)}$$

levels of logic.

Proof: Under the same assumptions as for theorem 4.1, at most $(\text{fan-in}-1)^L$ data inputs must be combined. For each original AND gate another AND gate is required, with the latch output replacing the \bar{C} input; these require half the fan-in-1 inputs to the OR gate. Thus, no more than

$$\log_{((\text{fan-in}-1)(\text{fan-in}-1)/2)}((\text{fan-in}-1)^L) = \frac{L \log(\text{fan-in}-1)}{\log((\text{fan-in}-1)^2/2)} \quad (4.22)$$

stages, or

$$\frac{2L \log(\text{fan-in}-1)}{\log((\text{fan-in}-1)^2/2)} \quad (4.23)$$

levels of logic are required. ■

For a fan-in of five, the length of a pipeline would increase to as much as 1.33 times the original length, if Earle latches are used instead of polarity-hold latches. Pipelines with polarity-hold latches require more care in the design of the clock circuits than do pipelines with Earle latches, but their lower latencies and gate counts outweigh this disadvantage. For this reason, polarity-hold latches are used almost universally in modern supercomputers.

4.3.2.2. Tuning Pipeline Parameters to Workloads

The best single choice of the number of pipeline stages S depends on the distribution of parallelism of the workload. Assume a hypothetical workload that has partitions with instances of groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 2048 operations of some type. Assume further that each value of N is equally important; that is, for every partition with 2048 operations of some type, there are two with 1024 partitions, four with 512 partitions, etc.

Table 4.5 shows the pipeline characteristics for several values of S . Only those values of S for which $pS - L$ is small are included; if $pS - L$ is too large, the pipeline latency is disproportionately large.

Table 4.6 shows the processing times for the various quantities of data N and for various numbers of pipeline stages S .

Table 4.7 shows the processing times for the various values of N and S , relative to the shortest processing time for each value of N . Small groups of operations are processed fastest by short pipelines, while large groups of operations are processed fastest by long pipelines.

Table 4.6: Processing Times for Various Quantities of Data N and for Various Numbers of Stages S .

Stages	Number of Data											
	1	2	4	8	16	32	64	128	256	512	1024	2048
2	73	97	146	244	439	829	1610	3172	6295	12541	25034	50020
3	75	94	131	206	357	658	1259	2462	4869	9682	19307	38558
4	80	96	128	192	320	576	1088	2112	4160	8256	16448	32832
5	83	97	125	180	291	514	959	1848	3627	7186	14303	28536
6	92	105	132	184	290	501	924	1768	3458	6837	13596	27112
9	111	122	144	188	277	455	810	1520	2941	5783	11466	22832
10	114	124	145	187	270	436	769	1435	2766	5428	10753	21403
13	135	145	164	203	281	436	746	1367	2609	5092	10058	19991
17	162	171	189	225	297	441	729	1305	2457	4761	9369	18585
25	215	224	240	273	340	473	738	1269	2332	4457	8706	17205
4'	83	100	133	199	332	598	1129	2191	4316	8566	17065	34063

Table 4.7: Processing Ratios for Various Quantities of Data N and for Various Numbers of Stages S .

Stages	Number of Data											
	1	2	4	8	16	32	64	128	256	512	1024	2048
2	1.00	1.03	1.17	1.36	1.63	1.90	2.21	2.50	2.70	2.81	2.88	2.91
3	1.03	1.00	1.05	1.14	1.32	1.51	1.73	1.94	2.09	2.17	2.22	2.24
4	1.10	1.02	1.02	1.07	1.19	1.32	1.49	1.66	1.78	1.85	1.89	1.91
5	1.14	1.03	1.00	1.00	1.08	1.18	1.32	1.46	1.56	1.61	1.64	1.66
6	1.26	1.12	1.06	1.02	1.07	1.15	1.27	1.39	1.48	1.53	1.56	1.58
9	1.52	1.30	1.15	1.04	1.03	1.04	1.11	1.20	1.26	1.30	1.32	1.33
10	1.56	1.32	1.16	1.04	1.00	1.00	1.05	1.13	1.19	1.22	1.24	1.24
13	1.85	1.54	1.31	1.13	1.04	1.00	1.02	1.08	1.12	1.14	1.16	1.16
17	2.22	1.82	1.51	1.25	1.10	1.01	1.00	1.03	1.05	1.07	1.08	1.08
25	2.95	2.38	1.92	1.52	1.26	1.08	1.01	1.00	1.00	1.00	1.00	1.00
4'	1.14	1.06	1.06	1.11	1.23	1.37	1.55	1.73	1.85	1.92	1.96	1.98

The weighted mean of the entries of a particular row is the measure of the relative "goodness" of a choice of pipeline length for the workload. The lower the mean, the closer that single pipeline approaches the performance of several pipelines, each used for the small range of operation group sizes for which it is optimal. The weighted means for the hypothetical workload are shown in table 4.8. A pipeline with ten stages, each with five levels of logic, is optimal, and is only 18% slower than a collection of pipelines of all the lengths.

Table 4.8: Weighted Mean Processing Ratios for Various Numbers of Stages, Assuming Uniformly Distributed Workload.

Stages	Mean Ratio
2	2.01
3	1.62
4	1.44
5	1.31
6	1.29
9	1.22
10	1.18
13	1.21
17	1.27
25	1.43
4'/25	1.06

The parallelism within Livermore Kernels 1-14⁶⁷ varies over a wide range; some partitions have only one floating-point operation, while others have as many as 1000 floating-point operations. The number of instances of groups of parallel floating-point operations of each size are tabulated in table 4.9.

Table 4.9: Numbers of Instances of Groups of Parallel Operations of Each Range of Sizes, with Weights.

Range	Number	Weight
1	5867	5.73
2	435	0.85
3-4	285	1.11
5-8	10	0.08
9-16	0	0.00
17-32	10	0.31
33-64	8	0.50
65-128	14	1.75
129-256	0	0.00
257-512	5	2.50
513-1024	3	3.00
1025-2048	0	0.00

The only floating-point operations in these kernels are adds, subtracts, and multiplies. Statistics for these are grouped together, although collected separately (thus a partition with 100 adds and 100 multiplies counts as two instances of groups of 100 operations). The weight of each range of parallel group sizes is the product of the group size and the number of instances, normalized by dividing by 1024. Thus, the importance of each instance of a group

is proportional to the number of operations that it represents. The distribution of weights indicate that overall performance of pipelined functional units depends on the performance for very small groups and the performance for relatively large groups of operations, in roughly equal proportion.

The weighted mean processing ratios for the workload distribution of Livermore Kernels 1-14 are shown in table 4.10.

Table 4.10: Weighted Mean Processing Ratios for Various Numbers of Stages S , Assuming the Workload Distribution of Livermore Kernels 1-14.

Stages	Mean Weighted Ratio
2	1.88
3	1.57
4	1.44
5	1.33
6	1.35
9	1.35
10	1.33
13	1.42
17	1.55
25	1.85
4'/25	1.06

The best performance is again for $S = 10$, but it is now 33% slower than a collection of pipelines of all lengths. This is due to the concentration of weights towards the extremes of group sizes.

4.3.2.3. Replicated Pipelines

Much better performance with the Livermore Kernels workload can be had with two separate pipelines for each operation. Full-speed pipelines with $S = 25$ and running at an 8.3-gate-delay clock can execute the parallel operations fast, while half-speed pipelines with $S = 4$ and running at a 16.6-gate-delay clock can execute the serial operations fast. Tables 4.6 and 4.7 list processing times and processing ratios for the half-speed pipeline as the entry with $S = 4'$.

The full-speed pipeline should be used for groups of operations larger than 16, while the half-speed pipeline should be used for groups of operations between one and 16 in size. Weighted mean processing ratios for this pair of pipelines are shown in tables 4.8 and 4.10 as the entries with $S = 4^{1/25}$.

The weighted mean processing ratio for both workloads are both 1.06, superior to the weighted mean processing ratio of any single pipeline for either workload. The cost of this performance is twice the pipeline hardware plus increased complexity of the functional unit routing networks.

Another alternative is to use several identical pipelines to increase throughput without increasing latency. If the number of pipelines for each function is increased from one to D , the total processing time is reduced to $T_D = \left[\frac{N}{D} + S \right] (C_{\text{high}} + C_{\text{low}})$. The optimal choice for p is increased by a factor of $(D)^{1/2}$:

$$p = \max \left[2, \left[\frac{DLU_{C_{i-1}, C_i}}{Nt_{\text{max}}} \right]^{1/2}, \left[\frac{2DL((2-r)t_{\text{max}} + U_{C_{i-1}, C_i} + U_{C, C})}{N(1-r)t_{\text{max}}} \right]^{1/2} \right]. \quad (4.24)$$

Of course, the hardware cost is increased by a factor of D plus the cost to route data to and from the pipelines, which grows by a factor of $\log_{\text{far-in}}(D)$.

4.3.3. Concessions to Slow Chip I/O

The bandwidth of chip I/O pins is often considerably below that possible within the chip. If the pipeline clock rate is limited to the chip I/O rate, the pipeline bandwidth is low. Under these circumstances, the number of stages S should be small, so that pipeline latency is low. Pipeline bandwidth can only be increased through duplication of pipelines, as discussed in the previous section.

Chip I/O bandwidth requirements can be reduced by storing some values in registers on the chip. Some operations, such as multiplying a vector by a scalar, lend themselves to this

approach. Depending on the degree of integration available on each chip, it may be possible to store parts of matrices on each chip. Some algorithms, such as parallel conjugate gradient methods for solving LU factorization problems, have clusters of computations on limited sets of data with limited communication among clusters.⁷³

As an alternative to implementations with slow clocks and few stages, nonlinear pipelines can be used. These have been used in such systems as the floating-point execution unit of the IBM System/360 Model 91,⁴ and have been extensively discussed by Davidson, Kogge, Shar, and Thomas.^{52,88,98} Nonlinear pipelines use some pipeline stages more than once for each operation, so operations cannot start each clock tick. They have the advantage of greater hardware efficiency, relative to the linear pipelines used in most modern computers, when new operations cannot be started every clock tick.

Nonlinear pipelines could be implemented with internal clocks running several times faster than the rate at which data are transferred and new operations start. They would be able to take advantage of the ratio of on-chip to off-chip speed, and their re-use of pipeline stages would provide savings in chip area, so that more complex functions could be implemented on a chip. The hardware savings would allow more pipelines to be included in the CPU, increasing the execution bandwidth.

4.4. Summary of Datapath-Design Properties

- (1): Propagation distances grow at a rate between the square root of the number of devices and linearly with the number of devices in a plane. Propagation distances grow at a rate between the cube root of the number of devices and the square root of the number of devices in three-space.
- (2): With unbalanced fan-in trees, $\text{fan-in}^i - a$ signals can be selected after i gate delays, $a \text{ fan-in}^j - b$ signals can be selected after $i + j$ gate delays, and so on. With unbalanced fan-out trees, $\text{fan-out}^i - a$ signals can be generated after i gate delays, $a \text{ fan-out}^j - b$

signals can be generated after $i + j$ gate delays, and so on.

- (3): Device decoding time is often negligible, relative to fan-out times or device multiplexing time.
- (4): The optimal number of levels of logic per pipeline stage is $\left[\frac{LU_{C_{i-1}C_i}}{Nt_{\max}} \right]^{1/2}$, where L is the number of levels of logic required to implement the function, $U_{C_{i-1}C_i}$ is the clock uncertainty, N is the number of independent sets of operands processed at a time, and t_{\max} is the maximum gate delay.
- (5): Variable-speed pipelines are not practical to implement, because the logic required to switch the number of pipeline stages increases the depth of the pipeline too much. For typical ECL gates, the length of the pipeline is increased by up to 55%.
- (6): A pipeline implemented using Earle latches requires up to 33% more levels of logic than a polarity-hold latch implementing the same function, for typical ECL gates.
- (7): With D identical pipelines, the optimal number of levels of logic per pipeline stage is

$$\left[\frac{DLU_{C_{i-1}C_i}}{Nt_{\max}} \right]^{1/2}$$

CHAPTER 5

Temporary Storage Devices and Access Mechanisms

The results presented in Chapter 3 establish the need for hundreds of temporary registers, and also show that most of the registers can have access times that are longer than one clock tick without a performance penalty. The results presented in Chapter 4 show how some devices in a set can be accessed faster than others. In this chapter, structures for temporary result storage are presented that provide both low latency and high bandwidth. Addressing and allocation issues are also discussed.

The temporary structures discussed include general-purpose registers, vector registers, and random-access-memory devices. Some of the general-purpose registers have fast access times and most of them have slower access times, but all can be accessed with high bandwidth. The vector registers support high-bandwidth access to all elements, with low-latency access to a small set of important elements. The random-access-memory devices provide high-bandwidth access to all locations, but do not necessarily provide low-latency access.

All of the structures discussed support many registers, and computers with many registers have long context-switch times. This is not a major disadvantage for supercomputers, however, because time-critical terminal and disk input and output operations are typically serviced by other computers dedicated to the tasks. In a time-sharing environment, the minimum running time quantum can be increased to the point where context-switch time is insignificant relative to running time. For procedure and function calls, most of the registers are not saved; the large number of registers allows different subsets of registers to be allocated to different procedures and functions. Of course, this philosophy complicates the register allocation process; however, if computational performance is considered less important than ease

of compiler construction or compiler execution time, register usage can always be restricted to a subset of the registers that are saved before every call.

5.1. Close/Distant General-Purpose Registers

The analyses summarized in Chapter 3 indicate that 256 registers are sufficient for fast execution of Livermore Kernels 1-14, as long as four to eight of them have one-tick access times. However, this requires considerable scheduling effort to achieve, and some of the kernels run longer when scheduled to minimize register usage rather than execution time, so more registers could be used to simplify the effort required to achieve fast execution. Consider the design of a set of 1024 general-purpose registers, a subset of which can be accessed in one clock tick. This subset of registers has capabilities similar to those of the Cray-1 and Cray X-MP, either of which can read two scalar registers, compute a logical combination of them, and write the result back to a scalar register in one clock tick using eight levels of logic.^{21,22,54}

The organization of such a set of registers is shown in figure 5.1. This register set can support up to two register reads and one register write each clock tick. At the left of figure 5.1 are 960 registers, organized as 30 sets of 32 registers each. A collection of 30 pairs of 32:1 multiplexers select the contents of any two registers. The multiplexer outputs are held in 30 pairs of latches controlled by the system clock. In the middle of the figure, 59 additional registers and the 30 pairs of latches feed a pair of 89:1 multiplexers, the outputs of which are held in a pair of latches controlled by the system clock. At the right of figure 5.1, five additional registers plus the pair of latches plus f other functional unit outputs are selected by a pair of $(7+f):1$ multiplexers. The outputs of these last two multiplexers feed a logical functional unit that performs operations like AND, OR, exclusive-OR, etc. The logical functional unit output feeds back to the five close registers, and is also held in a latch controlled by the system clock. The output of this latch is fanned out to the 59 middle registers and to the 960 distant

registers. In addition, a pair of 7:1 multiplexers can send the contents of any of the registers to the other functional units in the CPU.

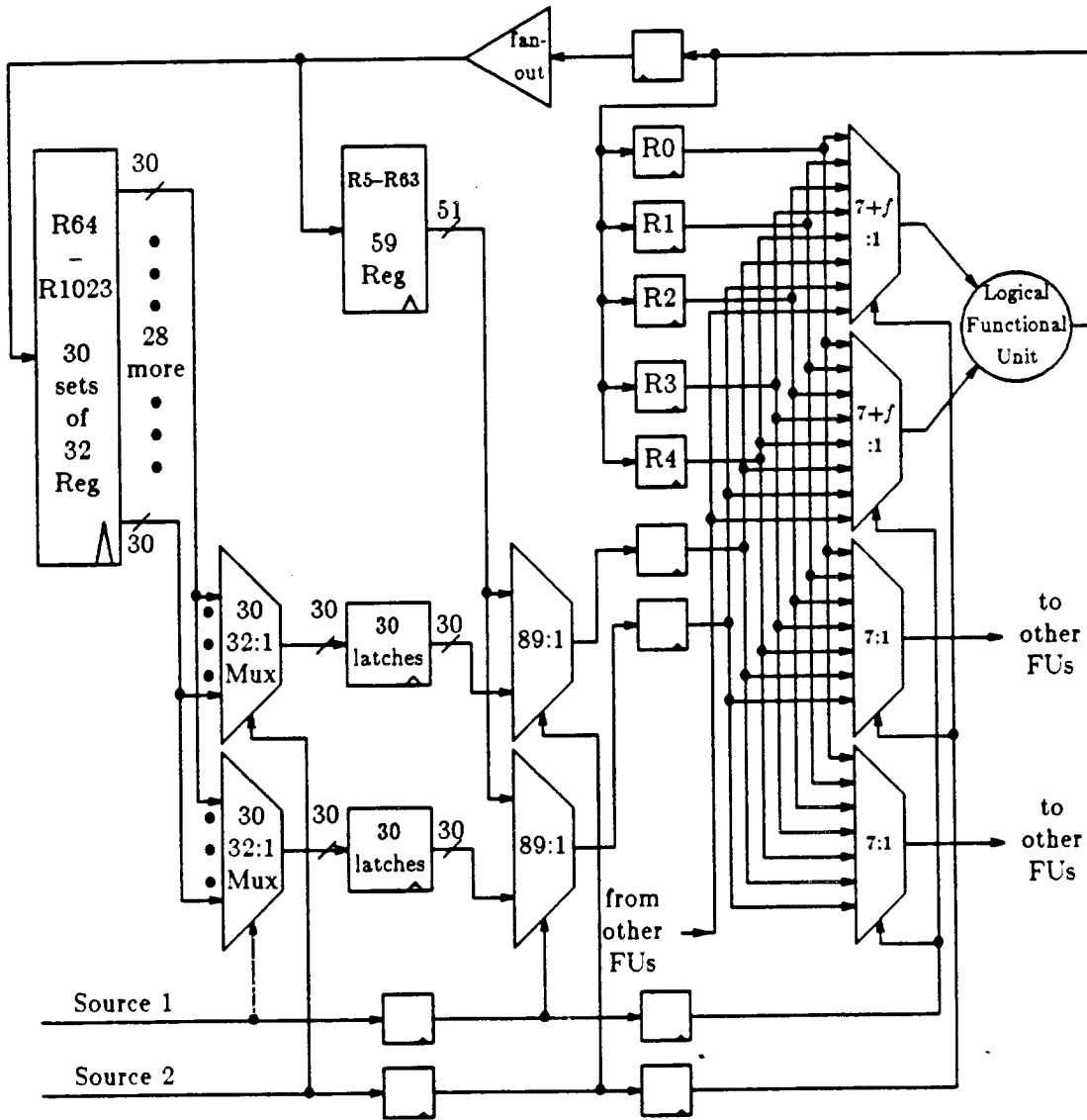


Figure 5.1: Organization of Close/Distant Registers.

Distant registers can be read and sent to the logical functional unit in two clock ticks, middle registers can be read and sent to the logical functional unit in one clock tick, and close registers can be read and sent to the logical functional unit in less than one clock tick. Read addresses reach the distant multiplexers directly from the operation start hardware, shifted six

bits toward the zeroes position. Read addresses reach the middle multiplexers after a one-tick delay from the operation start hardware, and are mapped into an appropriate form for the 89:1 multiplexers. Read addresses reach the close multiplexers after a two-tick delay from the operation start hardware, and are mapped into an appropriate form for the $(7+f):1$ multiplexers.

The output of the logical functional unit can be written to any close register in the same clock tick and to the middle and distant registers in the next tick. Decoded write addresses reach the close registers at the same time that the read addresses reach the close multiplexers. Decoded write addresses reach the middle and distant registers one tick after read addresses reach the close multiplexers.

Instructions with operation specifications must reach the start hardware at least two ticks before operations are actually started, in order to read distant operands in time. This increases the delay following a conditional branch before operations start again. This also forces any hardware reservation mechanisms to check the availability of registers as much as two ticks in the future, or longer if reservation checking takes more than one tick. With many close/distant registers there is, thus, a strong motivation to perform as much hardware reservation checking as possible at compile time.

5.1.1. Logic Design

Consider the logic required for a functional multiplexer that selects two of k registers and combines them logically. Each output bit can be expressed as a sum of products of register data, address bits, and function-control bits. Function-control bits and register-address bits are data independent, so they can be decoded down to as little as one bit for each product, one or more clock ticks before the logical operations are performed. Therefore the AND gates need not have more than three inputs each. With k registers and with logical functions of two inputs, there are $2k^2$ unique nonzero products. If the outputs of f other functional

units are input to the multiplexer, but are not combined with any of the register contents, f additional products of two terms are required. Thus, selecting two of k registers and combining them logically or selecting one of f functional unit outputs requires a sum of $2k^2$ products of three terms and f products of two terms.

Using gates with a fan-in of five, the products can be computed in one level of logic, but the sum requires $\lceil \log_5(2k^2 + f) \rceil$ levels of logic. For k between four and seven and f no larger than 16, the sum can be performed in three levels of logic, so a minimum of four gate delays are required to select from two of seven registers and combine them logically. This is a lower bound on the clock period to perform this function.

Each register can be implemented as a polarity hold latch if a level of AND gates for the clock inputs is inserted into the OR fan-in tree before the final OR gate. Although the clock signals could be ANDed with the data at the first level of AND gates, this would increase the uncertainty of delay from clock to output of each latch, and could result in an overall slower clock rate. Using the extra level of AND gates has the additional benefit that most of the logic can be shared among the various fast registers; the initial level of AND gates and the first two levels of OR gates are shared, while a level of AND gates and an OR gate are dedicated to each bit of each register. Figure 5.2 shows an implementation of this function for two registers and one other functional unit input using gates with a maximum fan-in of three.

Returning to the original example of the close/distant register set with the logical functional unit, the functions of selecting and modifying two of seven registers are implemented in one pipeline stage of five levels of logic. Note that this pipeline has three fewer levels of logic per stage than the Cray-1.

Using the analyses of section 4.3.1, with a ratio r of minimum to maximum gate propagation delays of 0.3, a pipeline with five levels of logic corresponds to a clock with a 10.4 gate-delay period; if r is 0.6, the clock can have a period of 6.8 gate delays. This is

approximately half the clock period of the Cray-1.⁵⁴

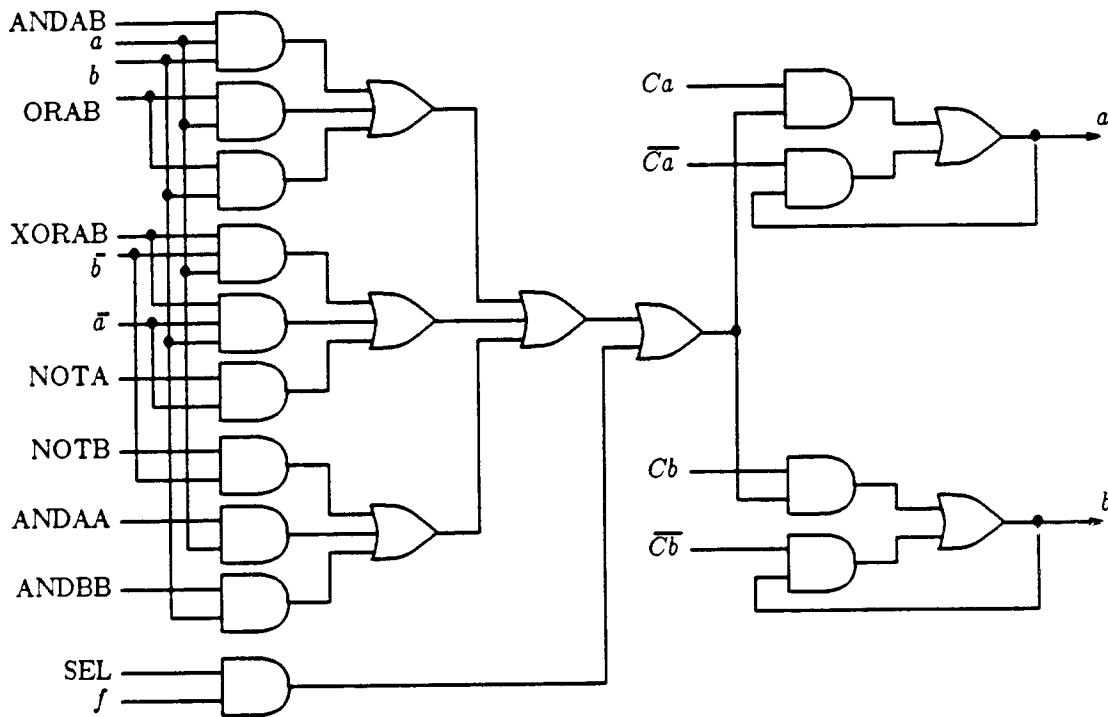


Figure 5.2: Implementation of a Polarity Hold Latch that Performs Selection and Logical Combination.

Five of the $7+f$ data inputs to the functional multiplexer are used for close registers, f data inputs are used for other functional units, and the remaining two inputs are used for middle and distant registers. The middle stage consists of 59 middle registers, a pair of 89:1 multiplexers, and a pair of latches. The 59 middle registers are implemented using polarity hold latches or other equally fast circuits.

Each multiplexer is directly connected to its latch, and both are implemented as five-level-of-logic polarity hold latches. The first three levels of logic consist of a level of AND gates for selection followed by two levels of OR gate fan-in. The last two levels of logic consist of a level of AND gates accepting data and clock inputs plus a single OR gate. With a fan-in of five, four data inputs plus the feedback input can be ORed by this last gate. The two-level OR fan-in trees can multiplex 25 signals down to one, so up to 100 inputs can be

selected and latched by either middle multiplexer-latch. Of the 89 data inputs used for each multiplexer-latch, 59 are used for middle registers and 30 are used for distant registers.

The distant stage consists of 960 distant registers organized as 30 groups of 32 registers each, 30 pairs of 32:1 multiplexers, and 30 pairs of latches. The distant registers are implemented using the same circuits as the middle registers. Each of the 60 distant multiplexer-latches are implemented as five-level-of-logic polarity hold latches that compute a 32:1 multiplexing function.

The bandwidth of the set of registers described is two reads and up to one write every clock tick, supporting a start-limit of one. Higher bandwidths supporting larger start-limits can be implemented, but increasing the bandwidth by b requires b^2 times as many product terms for the functional multiplexer. In effect, the functional multiplexer is replicated b times and each close register may be written with the output of any of the b functional unit outputs. This extra selection delay increases the number of levels of logic by $\log_{\text{fan-in}}(b^2)$, increasing the minimum clock period proportionately. Note that this increase in the clock period is necessary regardless of whether close/distant registers or traditional registers are used.

For close/distant registers where each functional multiplexer has c close register inputs, two middle and distant register inputs, and f other functional unit inputs, $2(b(c+2))^2 + f$ inputs must be multiplexed down to one and latched in one clock tick, where two levels of logic are taken up by AND gates. Thus, the clock must have a period long enough to accommodate at least $\left\lceil \log_{\text{fan-in}} \left[2b^2(c+2)^2 + f \right] \right\rceil + 2$ levels of logic. Assuming ECL's fan-in of five, up to 16 other functional unit inputs can be included without increasing the number of levels of logic. The numbers of levels of logic required for up to 16 other functional unit inputs and for various values of bandwidth b and numbers of close registers c are shown in table 5.1. In eight levels of logic as many as 16 close registers could be accessed with a start-limit of four, or as many as nine close registers could be accessed with a start-limit of eight.

Table 5.1: Numbers of Levels of Logic Required if All Inputs are Multiplexed Down to One Output, for Fan-in = 5, Up to 16 Other Functional Unit Inputs, with Various Numbers of Close Registers, and for Various Values of Bandwidth.

Read Bandwidth	Number of Close Registers														
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	5	5	5	5	6	6	6	6	6	6	6	6	6	6	7
2	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7
3	6	6	7	7	7	7	7	7	7	7	8	8	8	8	8
4	6	7	7	7	7	7	8	8	8	8	8	8	8	8	8
5	7	7	7	7	8	8	8	8	8	8	8	8	8	8	9
6	7	7	7	8	8	8	8	8	8	8	8	9	9	9	9
7	7	7	8	8	8	8	8	8	8	9	9	9	9	9	9
8	7	8	8	8	8	8	8	8	9	9	9	9	9	9	9

5.1.2. Close/Distant-Register Allocation

Close, middle, and distant registers should be allocated such that close registers are used for the most time-critical temporary results, middle registers are used for less time-critical temporary results, and distant registers are used for even less time-critical temporary results.

As long as an operation is separated from its source operations by at least

$$\text{start-limit} * (\text{operation-time} + \text{write-time} + \text{read-time})$$

other operations, it can start without any wait. (Note that even if data forwarding is used, the result must be written into some latch, and the destination operation must select from some number of source latches and registers, so a write time and a read time must still be included in the inter-operation separation.) Operations are allocated distant result-registers, and are scheduled so that they are separated from their source operations where possible. Based upon the studies summarized in Chapter 3, there should always be enough distant registers for all temporary results. For pairs of operations which cannot be scheduled far enough apart, the available middle registers are allocated to the more time-critical temporary results, and the available fast registers are allocated to the most time-critical temporary results.

Registers allocated for temporary results that are used by fewer operations are freed sooner than those allocated for temporary results that are read by more operations. Thus,

temporary results with shorter lifetimes should have higher priorities for middle and fast register allocation, other factors being equal. However, if the start-limit is greater than one, a temporary result that is used by several operations during the same clock tick should have higher priority than one used by fewer operations the same clock tick, because fast access to the former enables more operations to start.

The use of registers with different speeds undoubtedly complicates the task of automatic register allocation. However, the use of a large number of registers simplifies some aspects of register allocation, because there should always be enough registers available for temporary results. Therefore, the net effect may be only a slight increase in compiler complexity.

5.1.3. Instruction-Stream Bandwidth Considerations

A disadvantage of architectures with high instruction-stream-bandwidth requirements is that to achieve the same performance as architectures with lower instruction-stream-bandwidth requirements, they require larger instruction buffers; the larger instruction buffers cost more in terms of hardware, space, and power dissipation. Another disadvantage is that for computers with a single memory system for instructions and data, the instruction-stream interferes with the data stream to a greater extent. Fortunately, instruction-stream access is very regular and predictable for scientific programs, so it is possible to construct efficient instruction buffers with high performance. Higher instruction-stream bandwidth requirements can be tolerated if they lead to greater overall execution speed, but their costs in terms of hardware speed and performance are not negligible.

Architectures with many close/distant registers have longer register addresses than architectures with only a few registers, but their overall instruction bandwidth requirements are often less. Assuming seven-bit opcodes, a start-limit of one, and ten-bit close/distant addresses, a three-address instruction for close/distant registers has 37 bits. If small register sets have three-bit addresses, a three-address instruction for small register sets has only 16

bits. If memory addresses have 32 bits, a memory load or store instruction with a small register set has 42 bits, a memory load or store instruction with a close/distant register set has 49 bits, and a three-address memory-to-memory instruction has 103 bits. In addition to the reasons discussed in Chapter 3, pure memory-to-memory scalar architectures are poorly suited to scientific calculations because of their excessive requirements for instruction-stream bandwidth.

Each time the number of temporary results exceeds the number of registers in a small register set, a temporary result must be spilled, or written out to main memory, only to be loaded again before it can be used. Architectures with many close/distant registers require less instruction-stream bandwidth than architectures with small register sets whenever

$$37 * nonmem + 49 * mem < 16 * nonmem + 42 * mem + 42 * spill ,$$

where *nonmem* is the fraction of non-memory-referencing operations, *mem* is the fraction of memory-referencing operations, and *spill* is the ratio of temporary result spills and reloads to operations. Instruction-stream-bandwidth requirements for architectures with close/distant registers and for architectures with small register sets are shown in table 5.2 for Livermore Kernels 1-14. Except for the serial kernels 5, 6, and 11, the instruction-stream bandwidth requirements are higher for a small set of registers that spills than for a close/distant set of registers. Nevertheless, the bandwidth requirements are high, and it is desirable to reduce the instruction-stream bandwidth requirements of close/distant registers even further.

This can be accomplished for serial applications if a short instruction format is used when all operands are in close registers. For very-serial applications like Livermore Kernels 5, 6, and 11, all non-memory operands are in close registers, so close/distant addressing requires no more instruction-stream bandwidth than does small register set addressing.

Table 5.2: Instruction Stream Bandwidth Requirements in Bits per Operation for Close/Distant Registers and for a Set of 8 Registers, for Livermore Kernels 1-14.

Kernel	<i>nonmem</i>	<i>mem</i>	<i>spill</i>	Close/Distant BW	Small Set BW
1	0.62	0.38	1.61	41.56	93.19
2	0.45	0.55	1.38	43.60	88.05
3	0.50	0.50	1.47	43.00	90.73
4	0.50	0.50	1.43	43.00	89.05
5	0.40	0.60	0.00	44.20	31.60
6	0.40	0.60	0.00	44.20	31.60
7	0.80	0.20	1.78	39.70	96.06
8	0.70	0.30	1.67	40.60	94.01
9	0.61	0.39	1.58	41.68	92.73
10	0.31	0.69	1.28	45.28	87.69
11	0.33	0.67	0.00	45.04	33.34
12	0.33	0.67	1.32	45.04	88.90
13	0.33	0.67	1.46	45.04	94.55
14	0.54	0.46	0.45	42.52	46.97

5.2. Fast/Bulk Vector Registers

Vector instructions specify a series of identical operations to be performed on one or more vectors, producing a vector (or occasionally a scalar) result. Vector registers are used to hold the temporary results of vector operations; they can provide access to temporary results with lower latency and higher bandwidth than main memory.

Consider a vector register set that consists of eight vector registers, each containing 64 elements that have the machine's word size. In traditional usage, an entire vector register is addressed as a unit. When a vector register is read or written, the individual elements are accessed sequentially, one element per clock tick, starting with element zero. Usually only one sequence of element accesses can be in progress in each vector register at a time, either reading or writing; however, in some cases, a sequence of reads and a sequence of writes can be in progress in a vector register at the same time.

Traditionally, vector registers are implemented using many small fast random-access-memory (RAM) chips, as shown in figure 5.3.

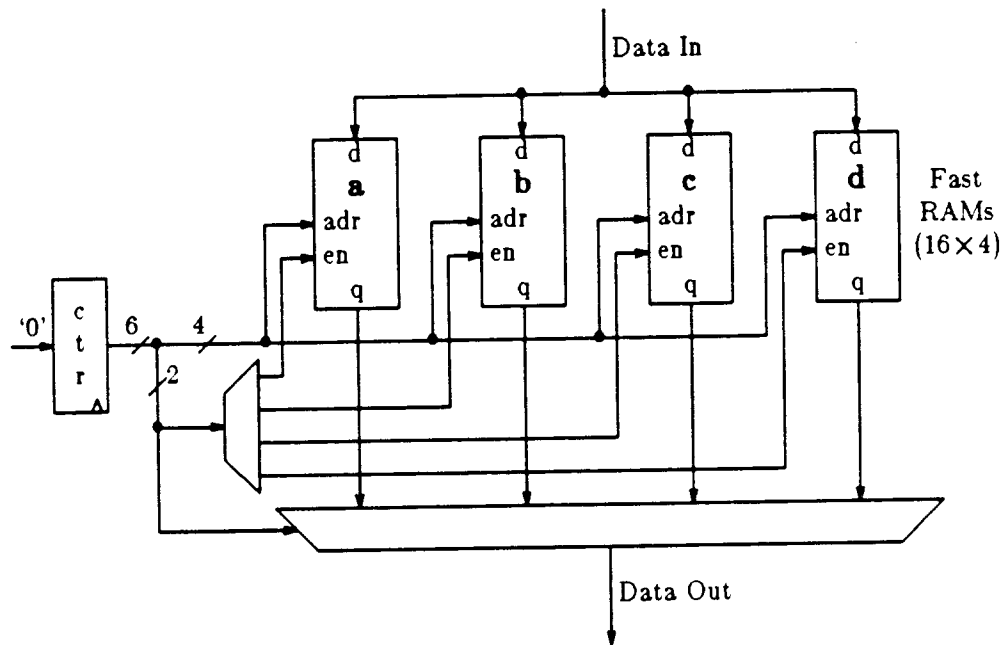


Figure 5.3: Implementation of a Traditional 64-Element Vector Register.

A six-bit counter generates element addresses, which are partially decoded and sent to four sets of 16×4 fast RAM chips. For 64-bit words each vector register uses 64 RAM chips, organized as four sets of 16 chips. When a vector element is read, the outputs of one set of RAM chips is selected and sent to a vector functional unit. When a vector element is written, the datum is distributed to the sets of RAM chips and one set is enabled for writing. The timing of a write followed by a read of a seven-element vector is shown in figure 5.4. Elements zero through six are written successively to RAM sets **a**, **b**, **c**, **d**, **a**, **b**, and **c**, starting at tick zero, one element per clock tick. Read timing is similar, and starts at tick seven.

The RAM chips described are fast enough to be accessed in one clock tick, so they are necessarily small, and many of them are required. However, vector access patterns are predictable, so RAMs this fast are not necessary.

RAM	Clock Tick													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a:	W_0				W_4			R_0				R_4		
b:		W_1				W_5			R_1				R_5	
c:			W_2				W_6			R_2				R_6
d:				W_3							R_3			

Figure 5.4: Vector Register Access Timing for a Write and a Read of a 7-Element Vector.

Element zero of a vector register must be accessed in the same clock tick that the vector access begins. Element one must be accessed by the tick following the one in which the vector access begins, and, in general, element j must be accessed no later than j ticks after the vector access begins.

The first few elements need low-latency access, but the remaining elements can tolerate a longer access latency if their access bandwidth is sufficiently high. Consider a fast/bulk vector register with a set of small fast RAMs for the first few elements, and two or more sets of slower bulk RAMs for the remaining elements, as shown in figure 5.5. The first two elements are stored in the set of fast RAMs, shown implemented using 16×4 RAM chips. These are addressed by a seven-bit counter, but are only enabled when addresses are less than two. The remaining elements are stored in the two sets of bulk RAMs, shown implemented using 64×4 RAM chips. These have a two-tick access time, are addressed by a separate seven-bit counter, and are only enabled when addresses are at least two. Address and data latches are used at the bulk-RAM inputs to hold them stable for the two-tick access times.

The timing of a write followed by a read of a seven-element vector is shown in figure 5.6 for this fast/bulk vector register. Vector elements are received from a vector functional unit in sequential order, so they must be written in that order. Elements zero and one are written to the fast-RAM set **a** one tick apart. Element two is written to bulk-RAM set **B** at tick two, and it keeps that RAM set busy through tick three. At tick three, element three is written to bulk-RAM set **C**, keeping it busy through tick four. By tick four, bulk-RAM set **B** is free so

element four can be written there. This interleaved write access continues until tick six, when element six is written to bulk RAM set **B**, keeping it busy through tick seven.

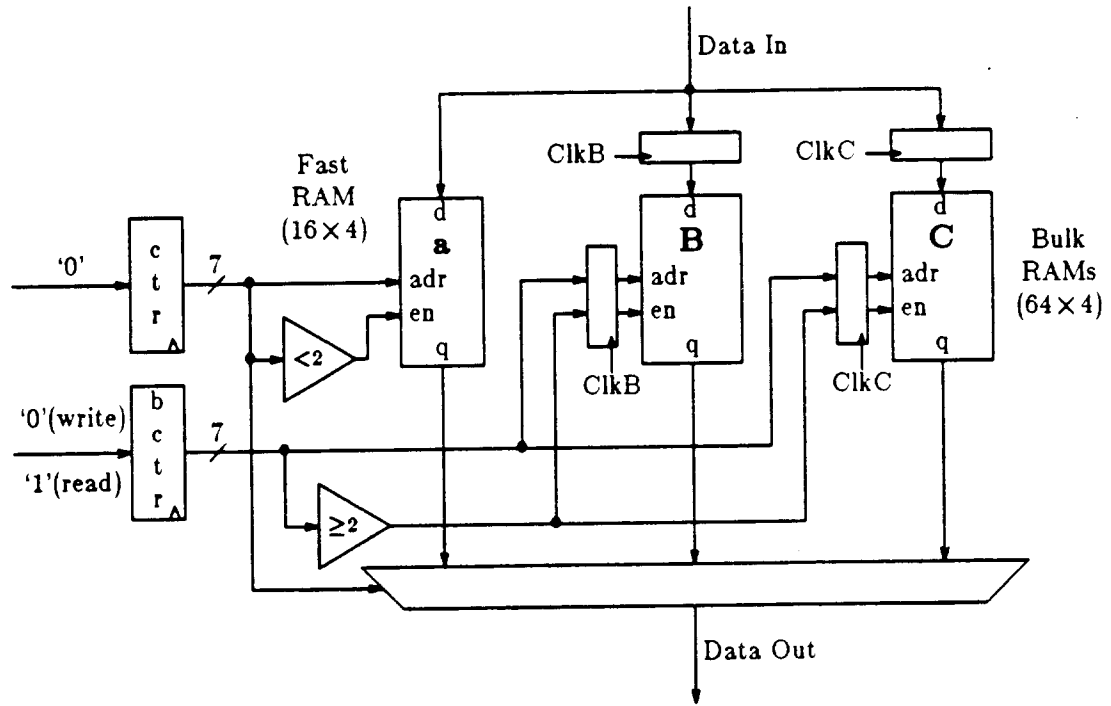


Figure 5.5: Implementation of a Fast/Bulk Vector Register with 128 Elements.

RAM	Clock Tick													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a:	W_0	W_1							R_0	R_1				
B:			W_2	\rightarrow	W_4	\rightarrow	W_6	\rightarrow	R_2	\rightarrow	R_4	\rightarrow	R_6	\rightarrow
C:				W_3	\rightarrow	W_5	\rightarrow			R_3	\rightarrow	R_5	\rightarrow	

Figure 5.6: Fast/Bulk Vector Register Access Timing for a Write Followed by a Read of a 7-Element Vector.

At tick seven, element zero is read from fast-RAM set **a**, and at tick eight, element one is read from fast-RAM set **a**. At the same time, the read access of element two begins from bulk-RAM set **B**; the read is completed at tick nine. The read access of element three begins at tick nine, and completes at tick ten. This interleaved read access continues until the read access of element six completes at tick thirteen.

Note that the external interface to the fast/bulk vector register is identical to that of the traditional vector register discussed before; the write access begins at tick zero, the first read access begins at tick seven, and the last read access completes at tick thirteen.

Bulk read accesses begin one tick before they complete to allow for their longer latency. For reads, therefore, the bulk address counter **bctr** is loaded with the constant one, while the fast address counter **ctr** is loaded with the constant zero; both **bctr** and **ctr** count together, so their outputs always differ by one. For writes, elements must be written in the order received, so **bctr** is loaded with the constant zero and the two counter outputs are the same. As the example shows, bulk-RAM set **B** can still be busy by the time a vector read begins, so the first two elements are stored in fast RAM, giving bulk-RAM set **B** time to recover after a write before it is pre-read.

Theorem 5.1: In general, with one access per clock tick, a fast-RAM access time of one tick, a bulk-RAM read access time of $t_{B\text{access}}$ ticks, a bulk-RAM busy time of $t_{B\text{busy}}$ ticks, and assuming a vector read can begin immediately after a vector write begins, at least $t_{B\text{busy}}$ sets of bulk RAM are required with at least $t_{B\text{busy}} + t_{B\text{access}} - 2$ elements in fast RAM.

Proof: Clearly $t_{B\text{busy}}$ sets of $t_{B\text{busy}}$ -tick-busy-time RAM are needed to keep up with an access bandwidth of one element per tick.

The first element in bulk RAM must be read $t_{B\text{access}} - 1$ ticks before it is needed, in order for it to be delivered on time. However, its bulk-RAM set could have been written as little as one tick before the vector read began, keeping it busy until $t_{B\text{busy}} - 1$ ticks into the vector read. Thus, in the worst case, a bulk-RAM element would not be delivered until $t_{B\text{busy}} + t_{B\text{access}} - 1$ ticks after the beginning of a vector read, so $t_{B\text{busy}} + t_{B\text{access}} - 2$ elements must be stored in fast RAM. ■

Corollary 5.2: If slow RAM sets with access time $t_{S\text{access}}$ ticks and with busy time $t_{S\text{busy}}$ are added to a fast/bulk vector register, at least $t_{S\text{busy}}$ sets of slow RAM are required, with at least $t_{S\text{busy}} + t_{S\text{access}} - 2$ elements stored in faster RAM.

Proof: The minimum number of slow RAM sets is determined by bandwidth considerations as it was for bulk-RAM sets in theorem 5.1.

Similar to the bulk-RAM case, the first slow-RAM element could be delivered as late as $t_{S\text{busy}} + t_{S\text{access}} - 1$ ticks after the beginning of a vector read, so $t_{S\text{busy}} + t_{S\text{access}} - 2$ elements must be stored in faster RAM. If $t_{B\text{busy}} + t_{B\text{access}} - 2$ elements are stored in fast RAM, bulk RAM must contain at least $t_{S\text{busy}} + t_{S\text{access}} - t_{B\text{busy}} - t_{B\text{access}}$ elements. ■

5.2.1. Advantages of Fast/Bulk Vector Registers

The advantages of fast/bulk vector registers over traditional vector registers are that fast/bulk vector registers allow longer vector registers for the same latency and bandwidth of access, they support lower latency and higher bandwidth for the same length vector registers, and they require fewer RAM chips for the same length and for the same latency and bandwidth of access.

The fast/bulk vector register shown in figure 5.5 has 128 elements, yet its external interface timing is identical to that of the traditional 64-element vector register shown in figure 5.3. All other factors being equal, RAM access time increases as the square root of the number of bits, because, for large enough RAMs, access time is dominated by physical signal propagation time across the two-dimensional array of bit cells. Thus, when the access time of the bulk RAMs is doubled, their size is quadrupled. If the bulk-RAM access time is significant relative to the clock period, doubling the access time requires doubling the number of sets of bulk RAM to maintain the same access bandwidth, so the net effect is to increase the size of

the vector registers by a factor of eight. Longer vector registers allow computations to run in less time because of fewer pipeline startup penalties. With vector registers of length 1024 or more, Livermore Kernel 3 could run with only one vector instruction per partition, and only one vector startup penalty per partition.

The fast elements of a fast/bulk vector register could be implemented using several registers, rather than fast RAM. In four levels of logic using ECL gates with a fan-in of five, a polarity hold latch implementing a 20:1 multiplexing function could be implemented. If there are no more than 20 fast element registers and bulk-RAM sets, a fast/bulk register could run at one half the latency and twice the bandwidth of the traditional vector registers of the Cray-1.

The fast/bulk vector register shown in figure 5.5 uses 16 fewer RAM chips than the traditional vector register of figure 5.3, for 64-bit words. Since quadrupling the size of the bulk RAMs doubles their access time, twice as many are needed to maintain the same bandwidth; however, for a long enough vector register, the change to larger bulk RAMs halves their number. The savings in RAM chips are partially offset by the extra address and data latches necessary to pipeline the vector accesses. The number of these extra latches is proportional to the number of sets of bulk RAMs.

The main reason for having several sets of bulk RAM is to increase the total bulk-RAM-access bandwidth. If a RAM chip existed that had high-bandwidth access, but not necessarily low-latency access, fast/bulk vector registers could be implemented very efficiently. An extra bulk-RAM counter would still be required in order to start bulk-RAM reads far enough in advance, but only one set of bulk-RAMs would be required. Furthermore, high bandwidth/higher latency RAM chips would eliminate the need for the address and data latches. Such a RAM chip is presented in section 5.3.

5.2.2. Vector-Access Extensions

5.2.2.1. Parallel Vector Access

Vector registers with bandwidths in excess of one element per clock tick can be implemented as traditional or fast/bulk vector registers with word sizes that are multiples of the machine word size. For example, a vector register with a bandwidth of four elements per clock tick for a computer with a 64-bit word size would have a word size of 256 bits, four times the machine word size.

During each clock tick, four parallel elements would be sent to four replicated pipelined functional units and four parallel results would be written to four parallel destination elements. Vector element counters would count by four, rather than by one; this would be accomplished by shifting the output of the original counter two bit positions away from the one's position. For vector lengths that are not multiples of four, extra operations would be performed at the ends of vectors, but this would impose no time penalty with parallel functional units.

Memory accesses would also proceed in parallel, four elements at a time for this example. The memory system would be responsible for inhibiting writing to extra memory locations if vector lengths were not multiples of four.

5.2.2.2. Simultaneous Read and Write Access to a Vector Register

On the one hand, the Cray-1 vector registers and the fast/bulk vector registers described above restrict access to either a vector read or a vector write, but not both at the same time. Furthermore, elements must be accessed at a steady rate, one per clock tick. On the other hand, each Cray X-MP vector register allows a vector write and a vector read to be in progress at the same time, so that each element can be read as soon as it is written, and the access rate need not be steady.

The Cray X-MP vector registers are frequently used as FIFOs between functional units, buffering results between chained vector operations. However, they differ from pure FIFOs because they can be read an arbitrary number of times after they are written.

Fast/bulk vector registers can be generalized to support this fast FIFO-like access, providing comparable access speedups and hardware savings. This is done by supplementing a set of bulk-RAM banks with a fast shift-register that buffers data between the write input and the read output. A generalized fast/bulk vector register is shown in figure 5.7.

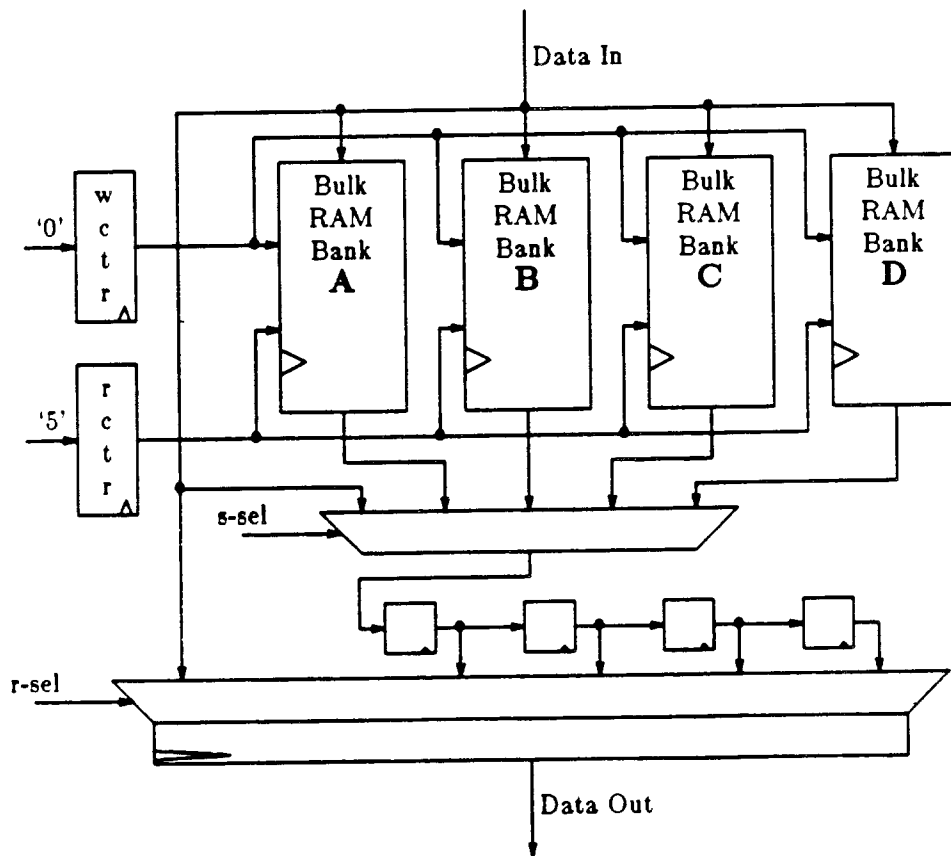


Figure 5.7: Generalized Fast/Bulk Vector Register Supporting Simultaneous Read and Write Access.

The bulk-RAM banks have busy times of two ticks, so four banks are necessary to support an access bandwidth of a read and a write per tick. Separate address counters allow read and write accesses to proceed independently. The output of the generalized fast/bulk vector

register can be selected from the data input itself or from the outputs of the shift-register.

Initially the write counter is set to zero, and the read counter is set to one more than the bank-busy time plus the bank access time; in this case it is set to five. The corresponding counter is incremented after each access.

As each element is written, it can be forwarded directly to the output latch or to the shift-register, depending on the difference between the write address and the read address. If the actual read address is equal to the actual write address (that is, if $rctr = wctr + 5$), then the element being written is copied to the output latch. If the difference between the actual addresses is one or two, the new element is shifted into the shift-register. If the difference is larger, the element is written into its RAM bank, but is not copied to any latch.

As each element is read, several events occur. The desired element is always available in the output latch, so it is sent to its destination. Unless the difference between the actual read and write addresses prior to the read was one, the older unread element of the shift-register is copied to the output latch. If the difference was five or more, a bulk-RAM bank read is initiated; when this completes the element is shifted into the shift-register.

When the read counter reaches the current vector length, it is reset to zero. The five final read accesses then initiate bulk RAM bank reads that prime the shift-register and output latches with the first five vector elements, so subsequent vector reads can start without delay.

Note that the shift-register shown uses one more shift-register cell than is absolutely necessary. Without it, the multiplexer before the output latch would need inputs for all the bulk-RAM banks, as well as for the data input and the shift-register outputs. The extra inputs might require more levels of logic in the multiplexer, which could require a longer clock period.

In general, there must be twice the number of banks as the bank-busy time to support the access bandwidth of a read and a write each tick. The shift-register must have as many

elements as the sum of the busy time and the access time of the bulk RAMs, in order to hold enough data for the first few reads.

Internal details of a bulk-RAM bank are shown in figure 5.8.

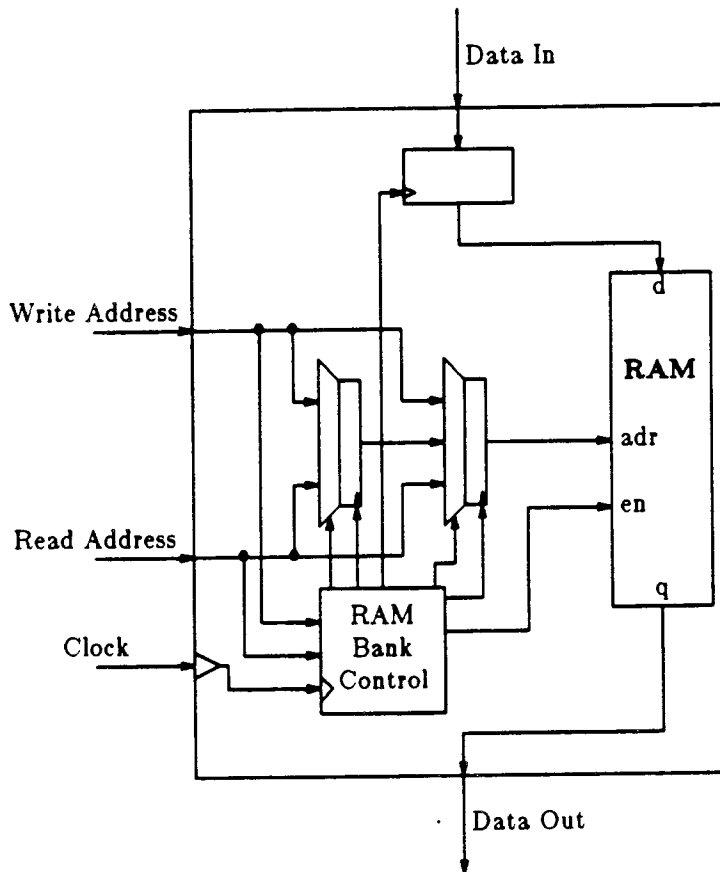


Figure 5.8: Internal Details of a Bulk-RAM Bank.

The bulk-RAM bank consists of bulk RAM, latches to hold data and addresses during the multi-tick access times, and some control circuitry. It is essentially the same as an interleaved main memory bank.

If the bank is free and a write is requested, the control circuitry recognizes the bank address, the upper bits of the address are copied to the right address latch, the input data are copied to the top latch, and the write access begins. If a read access to the bank is requested the following tick, before the bulk RAM is free, the read address is copied to the left address

latch and a read access begins the following tick. If both a read access and a write access are requested at the same time, the data input is copied to the top latch, one address is copied to the right latch and an access begins, while the other address is copied to the left latch and its access is deferred for the bulk-RAM busy-time.

Because vector read and write accesses are sequential, and because the bulk-RAM banks are interleaved, it is impossible for any bank to be accessed more frequently than once for a read and once for a write in the interval of twice the busy time. Thus, two address buffers and one data buffer for each bank are sufficient, regardless of the bank-busy time or the bank read-access time.

With the constraints that read accesses to elements must follow write accesses and that no more than one read access can start each tick, the write data forwarding and pre-reading insures that data are always available for read accesses. The latency of access for generalized fast/bulk vector registers is, therefore, one tick, regardless of the vector register's length.

The generalized fast/bulk vector registers can be extended to support one vector write and two vector read accesses simultaneously. This is done by increasing the interleaving depth from twice the bank-busy time to three times the bank-busy time; it requires adding another read counter, a shift multiplexer, a shift-register, an output multiplexer and an output latch for the second read access, and adding an extra address latch to each bulk-RAM bank. This access is more general than any access supported by the Cray-1,²¹ the Cray X-MP,²² the Cray-2,²³ and the NEC SX-2.⁴⁰ This extension can be applied to an arbitrary number of simultaneous vector reads by adding more RAM banks, multiplexers, shift-registers, and latches.

5.3. Pipelined Random-Access-Memory Chips

The fast/bulk vector registers discussed in the previous section require several banks of RAMs to achieve the necessary access bandwidth. The latency of access is not as important

as the bandwidth, however, because the fast registers can hide the latency. What is needed is a random-access-memory chip supporting high-bandwidth access, but not necessarily low-latency access.

Access to random-access-memory chips can be pipelined, allowing higher-bandwidth access than similar chips implemented without pipelined access. These pipelined RAM chips, or PRAM chips, are well suited for temporary storage for vector operations, particularly when used in a fast/bulk vector register implementation.

5.3.1. Memory-Access Delays

The sequence of delays associated with RAM chip read access is shown in figure 5.9.

- (1) receive address
- (2) decode address
- (3) propagate selects
- (4) enable bit cell output
- (5) propagate data
- (6) multiplex data
- (7) transmit datum

Figure 5.9: Sequence of Delays for Read Access.

Assume the RAM chip is organized with $2^k \times 1$ bits in a square array. On the one hand, select and data propagation times tend to dominate for larger RAM chips, because they grow as the square root of the number of bits. Decoding and multiplexing times, on the other hand, grow as the log of the square root of the number of bits, and address receiving and transmission times are independent of the number of bits. Data propagation times tend to be longer than select propagation times, because the many individual bit cells are usually smaller and less powerful than the few select line drivers.

The sequence of delays associated with RAM chip write accesses is shown in figure 5.10. This sequence is shorter than the read sequence because both addresses and datum are available at the same time and can propagate in parallel. Also, the bit cells only receive data, and do not drive long capacitive bit lines.

- (1) receive address and receive datum
- (2) decode address and fan out datum
- (3) propagate selects and propagate datum
- (4) write bit cell

Figure 5.10: Sequence of Delays for Write Access.

The read or write access latency is the sum of the delays along the respective sequences. For traditional RAM chips, the bandwidth is the reciprocal of the latency, as only one access can be in progress at a time. If one or more pipeline latches are inserted in the sequences, the bandwidth increases to the reciprocal of the latency of the longest delay between latches.

A limited degree of access pipelining has been applied to programmable-read-only-memory (PROM) chips, where a register is inserted before the output of the chip, allowing data to be used while a new word is read.⁶⁹

5.3.2. Limited Memory-Access Pipelining

Easy places to insert pipeline latches are at the output of the address decoders and write datum fan-out tree, and in the output of the read multiplexer. The address decoders and read multiplexers can be implemented as polarity hold latches.

A chip read is, thus, performed as a pipelined function of three stages: receive and decode; propagate selects, enable cell output, propagate data, and partially multiplex data; finish multiplexing data and transmit datum. A chip write is a pipelined function of two stages: receive address and datum, decode and fan out; propagate enables and datum and write cell.

The cost of this pipelined access is a clock input pin and the extra area required for the pipeline latches. Assuming four-or-six-transistor fast RAM cells are used, each RAM cell has roughly the same area as a gate. A polarity hold latch at the last stage of an address decoder can contribute only an AND gate to the function, so it costs two extra gates for each decoded output bit. Typically, half of the address bits are decoded and the other half are used to

control the select multiplexer, so $2 \cdot 2^{k/2}$ extra gates are required for decoder output latching for RAMs with 2^k bits. A polarity hold latch can perform a multiplexing function, so only the feedback AND gate is extra, and, therefore, multiplexer latching costs $2^{k/2}$ extra gates. A polarity hold latch can contribute only its OR gate to datum fan-out, so if one input fans out to $2^{k/2}$ data lines, the cost is at most $2 \cdot 2^{k/2}$ extra gates.

The total overhead of pipeline latches is, thus, $5 \cdot 2^{k/2}$ gates or bit cells. The overhead is shown in table 5.3 as a fraction of the 2^k bits stored in the RAM, as a function of the number of bits in the RAM.

Table 5.3: Fraction of Area Overhead Due to Pipeline Latches, as a Function of the Number of Bits.

Bits	Overhead
16	1.25
64	.63
256	.31
1024	.16
4096	.08
16384	.04
65536	.02

For small RAMs, the latch overhead is large, but the bit-cell area of these RAMs is relatively insignificant; the pin pad area often limits the minimum size of the chip. For RAMs larger than 1024 bits, the latch overhead is less than 10%, and the increase in access bandwidth easily justifies this cost.

This limited degree of pipelining has been implemented by GigaBit Logic in their 12G014 256 \times 4 pipelined static RAM.^{43,45} This RAM has latches at the data, control, and address inputs, and at the data outputs, and is able to operate with a 2.5 nanosecond cycle time. The busy time of this RAM is, thus, 2.5 nanoseconds, while the access time is 3.5 nanoseconds. The output latches can also be forced into a transparent mode, so that a slightly shorter access time can be achieved at the expense of a cycle time that is as long as the access time. It is possible to pipeline RAM chips to a greater degree than GigaBit does;

this is discussed in the following section.

5.3.3. Signal-Propagation Pipelining

For larger RAMs, physical signal propagation can be pipelined as well. The chip area devoted to bit-cells is partitioned into a small number of regions, with pipeline latches between regions. Figure 5.11 shows the placement of pipelined propagation latches for reads and writes for an eight-region chip, each region containing one eighth of the chip's bit-cells. Data and address lines within each region are distributed to all the bit-cells in the region. The pipeline latches are placed so that any path from address decoder to chip region to data multiplexer encounters the same number of latches. Also, the path from data input to any particular chip region encounters the same number of latches as the path from address input to that same chip region, so addresses match their data without requiring external skewing.

Regions are located in rows 0 through $n-1$ and columns 0 through $m-1$. Address inputs are conceptually located to the left of the top row at $(0, -1)$, data inputs are conceptually located above the left column at $(-1, 0)$, and data outputs are conceptually located below the right column at $(n, m-1)$. For any row of regions, a signal traveling from column x_i to column x_j must pass through $j-i$ pipeline latches, and for any column of regions, a signal traveling from row y_i to row y_j must pass through $j-i$ pipeline latches. The delay from an address or data input to region (x, y) is, thus, $x+y+1$ stages, and the delay from an address input to data output is $n+m$ stages.

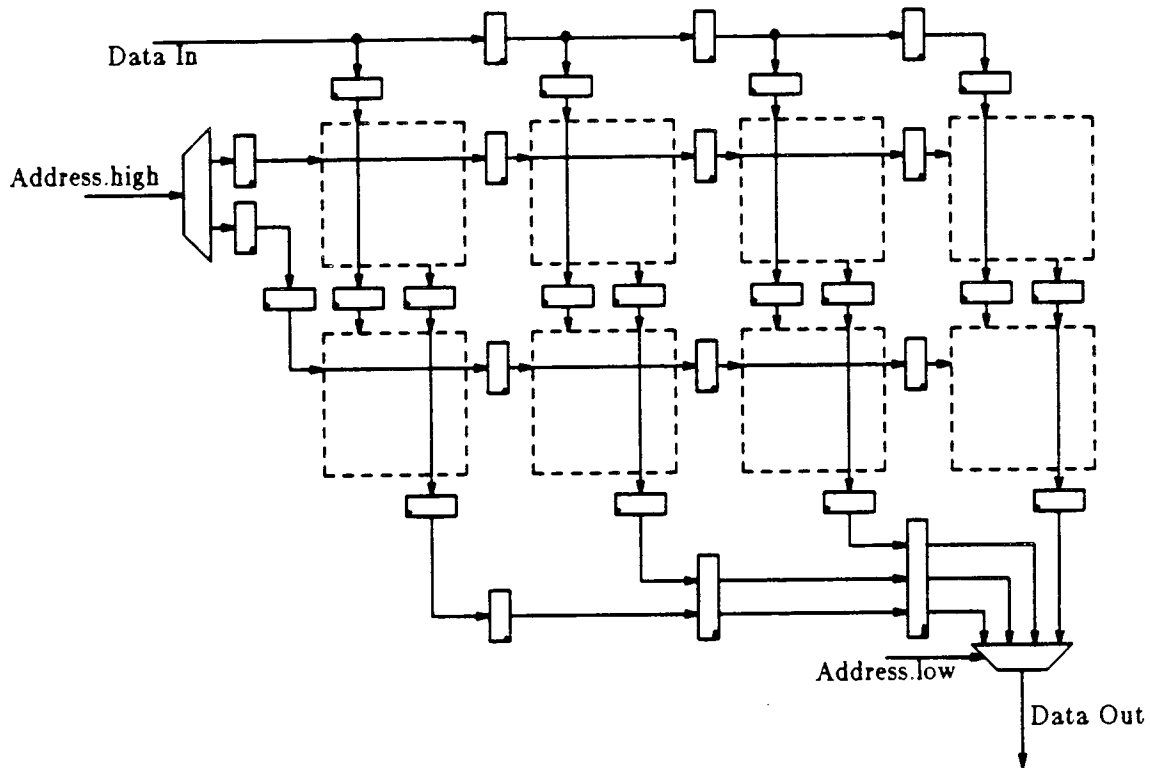


Figure 5.11: Placement of Pipelined-Propagation Latches for an Eight-Region Pipelined RAM Chip.

Pipeline RAM allows several accesses to be in progress at the same time, so there is a potential danger that a read access following a write access to the same location will access old data instead of the new data. Similarly, there are potential dangers of write-after-read conflicts and write-after-write conflicts. All these conflicts are caused by reads and writes occurring out of sequence.

Theorem 5.3: There can be no read-after-write, write-after-read, or write-after-write access conflicts with any pipelined RAM.

Proof: None of these conflicts can occur because the delay from address inputs to any location is a function of the region (and hence of the location) only, so all accesses to that location must proceed in strict sequence. The delay from address inputs to data outputs is constant for all regions, so read data

appears at the output in the same sequence that their addresses appeared at the inputs.

For pipelined RAMs without pipelined propagation, the numbers of latch delays to any location or from any location are independent of the location, so none of the conflicts can occur. ■

Read access delays are independent of location, so sequences of reads are delivered in proper sequence. In fact, read access delays are independent of the history of accesses and the bandwidth of access is access-pattern independent, so pipelined RAM is superior to interleaving.

This aggressively-pipelined RAM chip has three times the number of latches as the earlier example, so the latch overhead is proportionately higher. In addition, clock lines must be sent to latches in the middle of the chip, which costs additional chip area. For a 1024-bit RAM the overhead could be as high as 50%.

There are additional advantages to this aggressive pipelining scheme that partially mitigate the large overhead. First, the pipelined propagation reduces the length of the bit and select lines, and this in turn reduces the capacitance, resistance, and inductance proportionately; the combination of the reduced impedances decreases the propagation time by the square of the reduction in length, rather than linearly. Second, as each bit cell drives a smaller capacitance, it can be made smaller for the same speed, so more bit cells can be packed in the same area.

Despite these advantages, a PRAM chip will never have the density of traditional RAM chips, but they are intended for specialized high-performance applications, rather than for optimal storage density.

The latency of access for pipelined RAM chips is longer than for non-pipelined RAM chips because of the latch overhead. At the inputs and outputs of the RAM chip, the pipeline

latches implement necessary logical functions, so their contribution to the access latency is small. For propagation to and from the regions of bit cells, the latches increase the propagation delay by inserting gates in the propagation path; however, they also decrease the propagation delay by allowing the gates to drive lines with reduced impedances, as discussed above. The net effect is technology and process dependent, but each latch can be assumed to increase the propagation delay by at most two gate delays. Assuming that a non-pipelined RAM of the same size would have an access latency of 16 gate delays, a pipelined RAM with four regions and four pipeline stages would have an additional latency of eight gate delays, a 50% increase.

The effect of this increase in access latency is negligible for the intended applications; pipelined RAMs are used when access bandwidth is more important than access latency.

5.3.4. PRAM Applications

Pipelined random-access-memory is well suited to any application that traditionally makes use of interleaved random-access-memory and that does not require the shortest possible access latency. The property of access bandwidth that is independent of access pattern makes PRAM suitable for some applications with access patterns that cannot take advantage of interleaving.

An obvious application for PRAM is in the main memory system of a supercomputer, where storage density is traditionally sacrificed in favor of access bandwidth. Interleaving is still required to support simultaneous access of memory by one or more CPUs. However, memory banks implemented using PRAM have a busy time of at most one tick, as opposed to four to 60 ticks for traditional memory banks, so much less interleaving is required for the same performance.

There are no restrictions on access patterns, so arrays can be allocated and accessed without regard for performance implications. Also, accessing memory using a list of random

addresses can be as fast as using sequential access. Thus, the use of PRAM for main memory implementation should simplify algorithm development and code optimization.

Generalized fast/bulk vector registers can be implemented efficiently using PRAM. Assuming that small PRAM chips can run at twice the clock rate of the CPU, only a single bulk PRAM bank is necessary, as shown in figure 5.12.

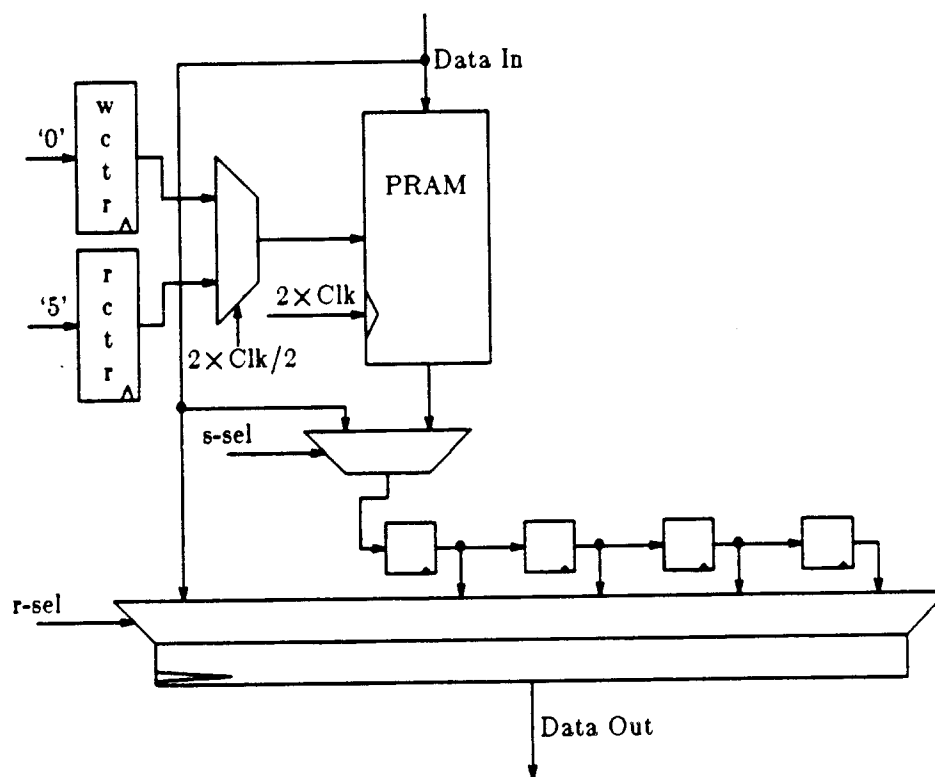


Figure 5.12: Fast/Bulk Vector Register with 64 Elements, Implemented Using Pipelined RAM Chips.

The PRAM chips have an access time of eight half ticks, and an effective busy time of zero because a read and a write access can be started each half tick. No additional latches for the PRAM bank are necessary because the PRAM chips are pipelined internally.

If the PRAM chips must be clocked at the same rate as the rest of the CPU, then two banks of them are necessary, with one additional address latch to hold simultaneously-arriving addresses. However, no other additional latches are required because of the internal pipelining

of PRAM chips.

If the PRAMs are implemented as 64×4 -bit chips, and assuming medium scale integration (MSI) logic chips, the generalized fast/bulk vector register in figure 5.12 uses as many chips as the traditional vector register in figure 5.3; the extra multiplexers and latches required by the fast/bulk vector registers use as many chips as the extra sets of fast RAMs in the traditional vector register. However, the fast/bulk vector register has twice the access bandwidth of the traditional vector register. Furthermore, if the lengths of the vector registers are increased, the chip count for the traditional vector register grows four times as fast as the chip count for the fast/bulk vector register; if the lengths of the vector registers are increased to 1024 elements, the fast/bulk vector register implemented using PRAM chips has one fourth the number of chips of the traditional vector register. Thus, fast/bulk vector registers implemented using PRAM chips can have twice the access bandwidth and can require one fourth the hardware required by traditional vector registers.

More general access patterns than sequential can be supported by fast/bulk vector registers if they are implemented using PRAM. For example, random-access reading and writing at full bandwidth is possible. This would allow loading and storing vector elements using out-of-order memory accesses, which might be required because of memory bank conflicts. Of course, more generalized vector access would require more complex address generation and checking circuitry.

CHAPTER 6

Analyses of Some Existing Supercomputers

6.1. Cray-1

6.1.1. Description

The Cray-1^{21,54,72,86} is a supercomputer in which all functions are pipelined with up to eight levels of logic and with a clock period of 12.5 nanoseconds. An instruction can be issued every clock tick. There are twelve independent functional units which perform shifts, logical operations, integer add, floating-point add, floating-point multiply, floating-point reciprocal approximation, population count, address add, and address multiply. Three of the functional units are dedicated to vector operations, three are shared between vector and scalar operations, four are dedicated to scalar operations, and two are dedicated to address operations. The machine has 64-bit data words and 24-bit addresses.

Vectors of up to 64 elements are supported in hardware, with eight vector-registers (V registers) of 64 elements each that can be accessed by vector instructions. During a vector read, one element is read from the V register every clock tick, for up to 64 ticks. During a vector write, one element is written to the V register each clock tick. Normally, only a single element can be read from or written to a vector register each tick, although in very restricted circumstances one element can be read from a V register and one element can be written to the same V register each tick. The eight V registers are independent, so during any clock tick a total of up to eight elements can be read from or written to the eight vector registers.

The results of vector instruction *a* cannot normally be used until all of its operations complete. However, if a vector instruction *b* follows *a* by no more than the chain slot time for

a , then b can use the results as they are written to the destination V register. This chaining allows b to start before all operations of a finish. No instruction can chain to a vector store.

There are eight scalar registers (S registers) of which any two can be read and any one can be written each clock tick. These scalar registers are supplemented with 64 intermediate storage registers (T registers). One word can be transferred between an S register and a T register each clock tick.

There are also eight address registers (A registers) of which any two can be read and any one can be written each clock tick. The address registers are supplemented with 64 intermediate storage registers (B registers). One word can be transferred between an A register and a B register each clock tick.

With a full configuration of memory, four million words are organized in 16 interleaved banks. Each bank has a four-tick busy time and an 11-tick load time. There is only one port connecting memory with the central processing unit (CPU) and input/output (I/O); that is, during any clock tick only one word can be transferred between memory and CPU or I/O.

6.1.2. Analysis

If the reported gate delay of .75 nanoseconds⁵⁴ is, in fact, the maximum gate delay t_{\max} , then the clock has a period of 16.7 gate delays; with eight levels of logic per pipeline stage, the clock period is 1.3 times the minimum predicted by the results presented in Chapter 4, so extra physical signal-propagation delays (beyond those necessary for correct pipeline operation) must add 30% to the clock period. This is consistent with the claim that physical signal-propagation delays account for half the total circuit delays.⁵⁴ Better packaging alone would, thus, appear to offer as much as a 30% improvement in performance.

The Cray-1 has a peak start-limit of eight. That is, it is possible for eight operations, each producing one result, to start execution each clock tick so that eight results can be pro-

duced each clock tick. This can be achieved by overlapping the execution of a vector load, six vector instructions using all six of the vector functional units, and a scalar instruction. For example, the code sequence shown in figure 6.1 produces nonsense results, but it serves to illustrate how overlapped vector instructions can start several operations each clock tick.

```

V0 ,A0, A1      ; load
V1 S1 *F V0     ; floating-point multiply
V2 S2 +F V1     ; floating-point add
V3 /H V2       ; floating-point reciprocal approximation
V4 S3 + V3      ; integer add
V5 S4 & V4      ; logical AND
V6 V5 < A2      ; left shift
S5 S5 + S4      ; scalar integer add

```

Figure 6.1: Code Sequence Producing a Start-Limit of Eight.

In the first instruction vector register V0 is loaded with 64 elements from memory locations at $(A0) + ((A1) * i)$, where i ranges from 0 to 63. Address register A0 is the base address, and address register A1 is the vector stride or interval between successive vector elements in memory. In the second instruction, every element of V0 is multiplied by the contents of scalar register S1, and the results are written to vector register V1. In the third instruction, the contents of S2 is added to every element of V1, and the sums are written to V2. In the fourth instruction, V3 is written with the half-precision reciprocals of the elements of V2. In the fifth instruction, V4 is written with the integer sums of the elements of V3 and S3. In the sixth instruction, V5 is written with the logical AND of V4 and S4. In the seventh instruction, V6 is written with the elements of V5, shifted left by the amount specified by address register A2. In the eighth instruction, S5 is written with the integer sum of S5 and S4. Operation timing is shown in figure 6.2.

The vector load instruction begins at tick 0, and a new load operation starts each clock tick (through tick 63). The first load operation finishes at tick 9, and its result is written to element zero of V0; this result is chained to the floating-point multiply instruction, which then begins execution. At this point, two operations are starting each tick: a load and a multiply.

The first product is chained to the floating-point add instruction at tick 18, which then begins execution. At this point, three operations start each tick: a load, a multiply, and an add. The sequence of chaining continues until the vector shift begins at tick 50. At this point, memory and all vector and floating-point functional units are busy, each starting a new operation every clock tick, for a total of seven operations starting each tick. A scalar integer add is started at tick 51, increasing the total of operations starting each tick to eight. Other scalar or address instructions could start in subsequent clock ticks, keeping the number of starts at the limit of eight through tick 63, when the last load of the first vector load instruction starts.

		Clock Tick																
		0	...	9	...	18	...	26	...	42	...	46	...	50	51	...	63	...
L_v																		
				$*F_v$														
					$+F_v$													
						$/H_v$												
								$+_v$										
										$\&_v$								
														$<_v$				
															$+_s$			

Figure 6.2: Operation Timing for Code Sequence Producing a Start-Limit of Eight.

The total start-limit is much smaller, in practice, because each parallel vector operation requires a separate functional unit, and the Cray-1 has only one of each type. Most real programs do not contain the diversity of operations found in the nonsense example of figure 6.1. The operation types found in each of the vectorizable Livermore Kernels are shown in table 6.1. Although they are vector operations, stores cannot be chained to, nor can they run overlapped with loads, so they do not contribute to the start-limit (kernels 3 and 4 each have only one scalar store, which could not significantly contribute to the total start-limit, even if stores could overlap with loads). The execution of the subtracts and adds in kernel 8 cannot overlap because they use the same functional unit.

Table 6.1: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the Cray-1.

Kernel	Operations	Actual Start-Limit
1	load, *, +, (store)	3
2	load, *, +, (store)	3
3	load, *, +	3
4	load, *, +	3
7	load, *, +, (store)	3
8	load, *, +, -, (store)	3
9	load, *, +, (store)	3
10	load, -, (store)	2
12	load, -, (store)	2

For scalar execution, the start-limit is at most one, because only one scalar operation can be started each tick, and some operations are delayed because of scalar result bus conflicts. For example, if a scalar shift instruction (which executes in two ticks) immediately follows a scalar integer add instruction (which executes in three ticks), they would finish at the same time; however, as only one scalar result can be written each tick, the shift instruction would stall for one or more ticks before issuing. A stalling instruction also prevents later instructions from issuing on the Cray-1. Result bus conflicts are not a serious problem, however, because they usually can be predicted at compile time, and instructions can be reordered to improve performance.

The Cray-1 would achieve higher start-limits and faster execution for the vectorizable kernels if it had replicated functional units or functional units that could perform more than one type of operation. For example, if the reciprocal-approximation functional unit could also perform multiplies, and if the integer add unit could also perform floating-point adds, then configuring the functional units such that there were two multipliers and two adders would double the maximum execution rate and nearly halve the execution time of the vectorizable kernels. However, the circuitry required to switch between functions would increase the latency of the functional units, and this would reduce performance for serial applications. Although more expensive than multifunctional units, replicated functional units offer increased

performance for parallel applications without significant degradation of serial applications, as discussed in Chapter 4.

The Cray-1 has 584 registers available for storing full-size temporary results, of which the 512 vector register elements and the eight scalar registers are readily available for use in computations. The T registers serve as a buffer between memory and the registers, but they cannot be accessed directly for operations; rather, they must be explicitly transferred to or from a scalar register using a separate instruction. The choice of 520 readily available registers corresponds well with the register usage analysis of Chapter 3, where all of the kernels run as fast with 256 registers as they do with an unlimited number of registers. However, kernel 14 needs as many as 128 registers to run within 10% of its unlimited-register speed, but it cannot be vectorized with current compiler technology. The problem with the Cray-1 is that scalar access of vector registers is very inefficient, so either the Cray-1 needs more general-purpose registers or more general access to vector registers.

The ratios of memory operations to all operations for Livermore Kernels 1–14, listed in table 5.2, range from 20% to 69% with an overall average of 53%. The memory bandwidth of the Cray-1 is one half of the combined bandwidths of the floating-point multiplier and adder, so memory can only keep up during the computationally-intensive kernels 1, 7, 8, and 9. Twice the Cray-1's memory bandwidth is necessary for memory to keep up with arithmetic in kernels 2, 3, 4, 10, 12, and 14, and four times the bandwidth is necessary for kernel 13. The memory bandwidth limitation of the Cray-1 is its most serious flaw.

6.1.3. Execution Rates

The simulator described in Chapter 3 has been modified to restrict the number of operations that can start each tick to the limit imposed by the functional units of the computer. For example, if there is one pipelined floating-point multiplier that can accept an operation each clock tick, only one multiply can start each clock tick. The execution rates for

Livermore Kernels 1-14 are computed using the times from the simulations and assuming a clock period of 12.5 nanoseconds. These are summarized in table 6.2.

Table 6.2: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Cray-1.

Kernel	Floating Point Ops	Ticks	MFLOPS
1	2000	1228	130
2	360	442	65.2
3	2048	2134	78.6
4	768	836	73.5
5	1992	12963	12.3
6	1994	12976	12.3
7	1920	1107	139
8	1440	878	131
9	1700	1231	110
10	900	2002	36.0
11	999	6009	13.3
12	1000	2003	39.9
13	1152	2486	37.1
14	1800	3794	38.0

These execution rates are higher than the fastest measured execution rates for these kernels, because the simulations ignore overhead operations and additional restrictions on datapath usage, such as functional unit recovery times or result bus conflicts. Nevertheless, the timings provide a uniform mechanism for comparing the performance of several different supercomputers.

6.2. Cray X-MP

6.2.1. Description

The Cray X-MP^{15,22,30,62} is a descendent of the Cray-1 with several important differences. The clock period has been reduced to 8.5 nanoseconds, and the single CPU of the Cray-1 has been replaced with four CPUs in the same physical space. Processors can communicate via shared memory or through a set of eight shared scalar registers, eight shared address registers, and 32 one-bit semaphore registers. A test and set instruction operates on

these semaphore registers, and provides the only hardware support for synchronization.

Memory has been expanded to eight million words, arranged in four sections of 16 banks of 128 kilowords each. The scalar load time is 14 clock ticks, up from 11 ticks in the Cray-1. Each CPU has four ports to memory: two for loads, one for stores, and one for I/O transfers.

The functional units are essentially the same as in the Cray-1, except that a second logical unit is housed in the floating-point multiplier, the use of which can be enabled in software. This can provide improved performance for programs with many logical operations but few floating-point multiply operations.

The vector registers can support an independent read operation and an independent write operation at the same time, which allows chaining to be generalized to work with any interval between source instruction and destination instruction.

There is no restriction on the number of scalar register results that can be written each clock tick, and there is no restriction on the number of address register results that can be written each clock tick. Result bus conflicts have been eliminated.

Vector gather-instructions and scatter-instructions have been added that allow loading or storing a vector indirectly through a vector of element addresses stored in a vector register; there are no restrictions on the addresses of the vector elements. During the execution of a vector gather-instruction or scatter-instruction, the elements are accessed sequentially, starting with element zero and ending with element $V_L - 1$, where V_L is the current vector length.

6.2.2. Analysis

The arithmetic-pipeline lengths for the Cray X-MP are the same as for the Cray-1 but the clock period is 32% shorter. It is possible that the denser packaging, that allows four CPUs and twice the memory to be packed into the same size space, also reduces interconnection lengths enough to reduce the clock period by 30%. That is, if the maximum gate delay

t_{\max} for the Cray X-MP is still .75 nanoseconds, then its pipelines are running at the theoretical maximum rate predicted by the results of Kunkel and Smith, summarized in Chapter 4.

The start-limit for each CPU of the Cray X-MP is larger than for the Cray-1 because two loads and a store can start each clock tick, instead of one load or one store, and operations can chain to stores. Table 6.3 shows actual start-limits for the vectorizable Livermore Kernels running on a single processor of the Cray X-MP. The start-limits increase from two or three to four or five.

Table 6.3: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the Cray X-MP.

Kernel	Operations	Actual Start-Limit
1	load, *, +, store	5
2	load, *, +, store	5
3	load, *, +	4
4	load, *, +	4
7	load, *, +, store	5
8	load, *, +, -, store	5
9	load, *, +, store	5
10	load, -, store	4
12	load, -, store	4

The second logical unit allows larger start-limits for computations involving many logical operations, without increasing the length of either the logical pipeline or the multiply pipeline. This is possible because only the inputs to the two functional units are shared—not the outputs or internal logic.

The limited memory bandwidth of the Cray-1 has been tripled in the Cray X-MP. For most of the Livermore Kernels the memory bandwidth is high enough to keep up with arithmetic operations, but for kernel 13 even higher bandwidth is necessary; the major flaw of the Cray-1 has been almost eliminated.

6.2.3. Execution Rates

The simulated execution rates for Livermore Kernels 1-14 on the Cray X-MP are summarized in table 6.4.

Table 6.4: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Cray X-MP.

Kernel	Floating Point Ops	Ticks	MFLOPS	Cray-1 MFLOPS
1	2000	1229	191	130
2	360	231	183	65.2
3	2048	1113	216	78.6
4	768	455	199	73.5
5	1992	12965	18.1	12.3
6	1994	12978	18.1	12.3
7	1920	1109	204	139
8	1440	865	196	131
9	1700	1080	185	110
10	900	1016	104	36.0
11	999	6011	19.6	13.3
12	1000	1022	115	39.9
13	1152	2878	47.1	37.1
14	1800	4249	49.8	38.0

The execution rates for many of the kernels are 50% faster than for the Cray-1 because of the 50% increase in the clock rate. However, kernels 2, 3, 4, 10, and 12 run almost three times faster on the Cray X-MP because of the additional memory bandwidth of the Cray X-MP. Kernels 13 and 14 run only 30% faster on the Cray X-MP because the memory latency of the Cray X-MP is only slightly less than that of the Cray-1.

The overall throughput of the four-processor Cray X-MP running independent computations is four times that listed above, because the memory system has enough bandwidth to support simultaneous accesses by all four processors.

6.3. Cray-2

6.3.1. Description

The Cray-2²³ is another descendent of the Cray-1, but there are many important differences. Four background processors and a single foreground processor are immersed in liquid fluorocarbon in an enclosure much smaller than that of the Cray-1 or Cray X-MP. The background processors execute programs and the foreground processor controls the system; only the background processors is discussed, and the following description applies to any one of the four background processors.

All functions are pipelined with a clock period of 4.1 nanoseconds in the Cray-2, but pipelines have more stages than the Cray-1 or the Cray X-MP have. Instruction issue takes two ticks, rather than one, so the scalar processing speed is not significantly faster than that of the Cray X-MP.

The Cray-2 has fewer functional units than does the Cray-1. Address add, address multiply, scalar shift, scalar logical, vector logical, and vector floating-point add perform essentially the same functions as the corresponding functional unit of the Cray-1. However, the scalar integer unit performs scalar integer add and subtract and scalar population count, and the vector integer unit performs vector integer add, subtract, vector population count, and vector shift operations. The floating-point multiply unit performs floating-point multiplies, reciprocal approximations, and reciprocal-square-root approximations. Timing information is scarce, but it appears that pipelines in the Cray-2 have twice as many stages as they do in the Cray-1 or the Cray X-MP.

The eight vector registers, each with 64 elements of 64-bit words, support no chaining at all. There are eight scalar registers and eight 32-bit address registers, but no B or T registers. These have been replaced by 16 kilowords of local memory, which has a four-tick access time. Vector transfers between local memory and vector registers can only have a stride of one; that is, successive elements accessed must be adjacent in local memory. In contrast, vector

transfers between main memory and vector registers can have any constant or variable stride on either the Cray X-MP or the Cray-2.

The Cray-2 has 256 million words of main memory shared among the five processors, organized in 128 interleaved banks. Memory has a 57-tick busy time and a 59-tick scalar load time. Each processor has only one port to main memory.

Vector gather-instructions and scatter-instructions similar to those in the Cray X-MP are also present in the Cray-2 architecture. All elements must be transferred between a vector register and memory in strict sequential order; in order to prevent some address sequences from overloading the memory bandwidth, gather-instructions and scatter-instructions operate at one fourth the normal rate, transferring just four elements every 16 ticks.

Synchronization among processors is supported by eight shared semaphore-bits.

6.3.2. Analysis

The Cray-2 has a peak start-limit of six, which can be achieved by keeping the memory port and the four vector functional units busy and then starting a scalar instruction. However, the lack of chaining makes it all but impossible to keep the four functional units busy computing useful results without running out of vector registers. Even if each arithmetic operation reads only one vector register and writes to another one, this only allows four vector operations to be in progress at a time, because vector registers cannot be shared by instructions unless chaining is supported. In the absence of chaining, eight vector registers are too few. The scalar start-limit for the Cray-2 is one half because scalar operations can only be issued every other clock tick.

The local memory is useful for temporary storage of vector results because it has a much faster access time than main memory, and instructions for transfers between vector registers and local memory are shorter than instructions for transfers between vector registers

and main memory. However, local memory cannot be used directly in vector operations, and, without chaining each transfer to and from local memory, it incurs a substantial time penalty. Local memory has a longer access time than the B and T registers of the Cray-1 and the Cray X-MP, but it does allow computed addresses to be used; B and T addresses are fixed in their instructions.

The memory latency of the Cray-2 is extremely long, leading to slow execution of memory-intensive applications like Livermore Kernels 13 and 14. The memory bandwidth with one processor accessing it is at least as poor as it is for the Cray-1 because each processor has only one memory port. In addition, the 57-tick busy time of each of the 128 main memory banks limits memory accesses to at most 2.25 accesses per clock tick, and a single processor could exceed the bandwidth of main memory with a vector load or store instruction with a stride of four. Even if they all accessed memory with only unit stride, just three processors could exceed the bandwidth of main memory. The main memory latency and bandwidth of the Cray-2 are entirely inadequate for the processors.

6.3.3. Execution Rates

The simulated execution rates of the Livermore Kernels running on the Cray-2 are listed in table 6.5. Parallel kernels 1, 7, 8, and 9 run almost twice as fast on the Cray-2 as they do on the Cray X-MP because of the clock rate of the Cray-2 is twice as fast. Serial kernels 5, 6, and 11 run slower because the functional unit pipelines of the Cray-2 are longer than those of the Cray X-MP, so their overhead is larger. Serial kernels 13 and 14 run slower because the memory latency of the Cray-2 is twice that of the Cray X-MP. The other kernels run approximately the same speed on the two machines, because the lack of adequate memory bandwidth on the Cray-2 nullifies the speedup due to higher computational bandwidth. These simulations do not make use of the Cray-2's local memory for storing arrays. If the arrays of kernels 13 and 14 could fit in local memory, performance on these kernels would increase to

87.3 MFLOPS and 119 MFLOPS, respectively, due to the reduced access latency of local memory.

Table 6.5: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Cray-2.

Kernel	Floating Point Ops	Ticks	MFLOPS	X-MP MFLOPS
1	2000	1299	367	191
2	360	497	172	183
3	2048	2325	210	216
4	768	991	185	199
5	1992	35919	13.2	18.1
6	1994	35955	13.2	18.1
7	1920	1178	388	204
8	1440	946	362	196
9	1700	1299	312	185
10	900	2002	107	104
11	999	18045	13.2	19.6
12	1000	2003	119	115
13	1152	10346	26.5	47.1
14	1800	14726	29.1	49.8

The Cray-2 has insufficient memory bandwidth even for one processor, so system throughput with all four background processors is considerably less than four times the throughput with one processor, at least for workloads with a moderate number of memory accesses. However, for programs with small working sets, the local memories of the processors can be used to reduce the main memory bandwidth requirements to some degree.

6.4. NEC SX-2

6.4.1. Description

The NEC SX-2^{40,49,50,104,106,111} is a supercomputer with a six nanosecond clock, 3.3 volt current-mode logic, and a gate delay of .25 nanoseconds. One instruction can be issued every clock tick. There are 20 pipelined functional units; the vector execution unit has four parallel multiply/divide functional units, four parallel add/subtract functional units, four parallel logical functional units, and four parallel shift functional units. All the parallel functional units of

a given type are used together. The scalar unit has a multiply/divide functional unit, an add/subtract functional unit, a logical functional unit, and a shift functional unit. Each functional unit can accept a new operation each clock tick, except for divides, where a new operation is accepted every second clock tick. Pipeline lengths are not published, but based upon analyses of published performance data for the Livermore Kernels, it appears that the multiply pipelines and add pipelines have no more than 11 stages each. The machine has 64-bit data words and 32-bit addresses, and supports 32-bit, 64-bit, and 128-bit floating-point formats.

There are 40 vector registers of 256 elements each, implemented using 3.5 nanosecond 256×4 RAM chips. The set of vector registers appears to support an access bandwidth of at least 16 vector element reads and 16 vector element writes each clock tick. Vector operations can be chained together. There are 128 scalar registers in the scalar execution unit.

The machine has a main memory of 32 million words, interleaved in 512 banks. Main memory has a busy time of at least seven clock ticks, and an access time of at least seven clock ticks. Vector loads access eight elements each clock tick, and vector stores access four elements each clock tick. All four or eight elements must be from the same vector instruction, and vector loads and stores cannot execute at the same time.

Scalar memory accesses are through a 64 kilobyte cache implemented using 3.5 nanosecond 256×4 RAM chips, but with otherwise unspecified parameters:

6.4.2. Analysis

With the reported gate delay of .25 nanoseconds and a clock period of six nanoseconds, the clock period is extremely long, allowing as many as 23 levels of logic in each pipeline stage. If the parallel pipelines were implemented with this many levels of logic, performance would be optimized for relatively short vectors. However, judging from the large physical size of the arithmetic processor (2.8 cubic meters for just the processor versus less than 1.4 cubic

meters for five processors, memory, and power supplies for the Cray-2), a large fraction of the pipeline delay is due to physical signal-propagation time, and scalar execution speed suffers as a result. Shorter physical paths between modules could allow a faster clock, which would produce equal vector performance at a lower cost than the replicated functional units.

The NEC SX-2 has a peak start-limit of 25. This can be achieved by starting a vector load, four vector arithmetic operations, and a scalar operation. Start-limits of 12 or 16 could be achieved for the vectorizable Livermore Kernels, as shown in table 6.6.

Table 6.6: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the NEC SX-2.

Kernel	Operations	Actual Start-Limit
1	load, *, +, (store)	16
2	load, *, +, (store)	16
3	load, *, +	16
4	load, *, +	16
7	load, *, +, (store)	16
8	load, *, +, -, (store)	16
9	load, *, +, (store)	16
10	load, -, (store)	12
12	load, -, (store)	12

Load operations can start eight at a time, and multiply and add operations can each start four at a time. The NEC SX-2 would achieve higher start-limits with these vectorizable kernels if vector loads and stores could execute at the same time. Also, supporting two different vector loads at the same time, as well as stores, would allow memory-to-memory vector operations to run without interruptions.

The NEC SX-2 has 10240 vector register elements, which is more than adequate for the applications analyzed in Chapter 3. A higher access bandwidth supporting a faster system clock would be possible using the fast/bulk vector registers described in Chapter 5.

The 128 scalar registers are adequate for most serial applications. The access time for 128 registers is necessarily long, but with such a slow clock speed (in terms of gate delays), they probably can be accessed within one (long) clock tick. Also, the instruction format has

32 bits to accommodate 7-bit register addresses, which requires twice the instruction stream bandwidth of the Cray X-MP.

Overall, the NEC SX-2 appears to be an extremely fast supercomputer, at least for parallel applications, but that speed is achieved using brute force, rather than through architectural innovation. With better organization and packaging, a computer with a faster clock and equal or better performance could be implemented using a fraction of the hardware.

6.4.3. Execution Rates

Simulated execution rates for the Livermore Kernels are shown in table 6.7. The enormous execution rates of the NEC SX-2 for parallel kernels are due to its parallel functional units that allow starting up to 16 operations each tick and to its fast clock rate (relative to the other computers). Performance for the serial applications is roughly the same as on the Cray X-MP, because both memory latency and pipeline lengths are assumed to be approximately the same.

In terms of overall system bandwidth, the four-processor Cray X-MP and the NEC SX-2 are much closer in performance; the Cray X-MP has more bandwidth for serial and moderately parallel workloads, while the NEC SX-2 has more bandwidth for very parallel workloads.

Table 6.7: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the NEC SX-2.

Kernel	Floating Point Ops	Ticks	MFLOPS	X-MP MFLOPS
1	2000	344	969	191
2	360	106	566	183
3	2048	410	833	216
4	768	217	590	199
5	1992	21935	15.1	18.1
6	1994	21957	15.1	18.1
7	1920	314	1019	204
8	1440	244	984	196
9	1700	304	932	185
10	900	272	552	104
11	999	11012	15.1	19.6
12	1000	283	589	115
13	1152	4318	44.5	47.1
14	1800	6678	44.9	49.8

6.5. HEP-1

6.5.1. Description

Although not fast enough to be considered a supercomputer, the HEP-1²⁶ has several interesting characteristics that warrant consideration. It is designed to support multiple instruction streams using a single processor, reminiscent of the Peripheral Processing Units of the Control Data 6600.^{99,100} Most of the functional units are pipelined with eight stages, so the basic execution time is eight 100 nanosecond clock ticks. Each clock tick an instruction from one of a set of parallel instruction streams is selected for execution. With eight or more instruction streams, an instruction can be selected and issued every clock tick, regardless of the instruction dependencies within each instruction stream.

Functional units include a floating-point add functional unit, a multiply functional unit, an integer functional unit, a divide functional unit, a scheduler functional unit, a hardware access functional unit, a process creation functional unit, and a system performance instrument. The floating-point add functional unit performs floating-point adds and subtracts and normalization. The multiply functional unit performs floating-point and integer multiplies.

The integer functional unit performs integer adds and subtracts, format conversion operations, logical operations, shifts, and compare operations. All these functional units have eight pipeline stages and can accept new operations each clock tick. The divide functional unit is not pipelined and executes in 17 clock ticks. It performs floating-point and integer divide operations. Up to eight divider functional units may be installed in a processor. The scheduler functional unit performs memory loads and stores. When the load time exceeds eight clock ticks, the scheduler functional unit notifies the destination instruction stream when the datum arrives. The hardware access functional unit is used to access I/O devices. The process creation functional unit provides a low-overhead mechanism for creating new instruction streams. The system performance instrument is used to keep track of process information for accounting purposes, but it can be used to measure performance as well.

There are 2048 registers that support two operand reads, one result write, and an extra read or write each clock tick. In addition, 4096 constant registers are available for read-only access by user processes.

Each register and data memory location has a three-state synchronization control mechanism, where the location can be marked empty, reserved, or full. This allows different instruction streams to communicate with very low overhead. The hardware simply requeues those instructions that unsuccessfully attempt to access a register or memory location, so that they automatically busy-wait.

6.5.2. Analysis

The HEP-1's most serious shortcoming is its clock period of 100 nanoseconds, which is surprising because ECL gates are used for its implementation. The primary reason for the long clock period is that four register accesses are made to 25 nanosecond RAM chips each clock tick. The clock period could be reduced by using pipelined RAM chips for the register implementation. Instruction issuing is pipelined on the HEP-1 so the register addresses are

known in advance. Thus, although register access bandwidth must be high, access latency can be relatively long. These characteristics are exactly what the pipelined RAM chips described in Chapter 5 provide.

Since most of the functional units have exactly eight pipeline stages, the instructions that use them can never cause result bus conflicts because, at most, one instruction is issued each clock tick. The divide functional unit and memory can cause result bus conflicts because they do not complete after exactly eight clock ticks. These operations are allocated a separate 25 nanosecond phase of every clock period on the chance that they may need to write a result. If a divide and a memory operation attempt to write results at the same time, one operation stalls.

Better use of the register set's bandwidth would be achieved by inhibiting the system clock every time a divide result or memory load operation writes to the register set and a write cycle is unavailable. This could allow the clock period to be reduced by 25%, and extra clock cycles, due to clock inhibition, would be inserted only when the extra register bandwidth is actually needed. If the clock is implemented with three phases, only one phase of one third the clock period would need to be inserted for extra writes.

6.5.3. Execution Rates

Simulated execution rates for the Livermore Kernels running on the HEP-1 are shown in table 6.8. The execution rates for all the kernels are a fraction of the rates for the Cray-1, because the clock rate is one eighth that of the Cray-1 and because the start-limit is one. Note that for many kernels, the HEP-1 executes in roughly as many ticks as the Cray-1 does; if the HEP-1 could be made to run with as fast a clock, it would be competitive.

Table 6.8: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the HEP-1.

Kernel	Floating Point Ops	Ticks	MFLOPS	X-MP MFLOPS
1	2000	1625	12.3	191
2	360	454	7.9	183
3	2048	2162	9.5	216
4	768	858	9.0	199
5	1992	15956	1.2	18.1
6	1994	15972	1.2	18.1
7	1920	1229	15.6	204
8	1440	1231	11.7	196
9	1700	1633	10.4	185
10	900	2002	4.5	104
11	999	8012	1.2	19.6
12	1000	2003	5.0	115
13	1152	3458	3.3	47.1
14	1800	3951	4.6	49.8

The system throughput of the HEP-1 is not significantly higher than the throughput for one instruction stream, because the parallelism present in all but kernels 5, 6, and 11 allows the HEP-1 to start some operation every clock tick; most kernels use the entire operation start bandwidth of the machine.

6.6. ETA-10

6.6.1. Description

The ETA-10^{33,34,84} is implemented using Advanced Large Scale Integration CMOS that is chilled in liquid nitrogen, and has a clock period of seven nanoseconds. The system can be configured with 64 to 256 megawords of shared memory, two to 18 input/output units, a one megaword communication buffer for transfers from I/O ports and CPUs, and two to eight CPUs. Data can be manipulated as bits, 8-bit bytes, 32-bit words, and 64-bit words. Data types include floating-point, fixed point, character, and decimal. Memory is accessed using virtual addresses.

Each CPU has a private memory, 256 registers, a scalar execution unit, and a vector execution unit. Private memory is four megawords in size, implemented as 16 banks of 32-bit

words. Each memory bank has an access time of 14 ticks and a busy time of seven ticks. The 256 registers are implemented as two 256-element register files, and two words can be read and one can be written each clock tick. Shortstop circuitry in the scalar datapath, similar to the data forwarding in the IBM System/360 Model 91,¹⁰³ allows results to be routed directly from the output of a functional unit to the input of another functional unit, reducing the temporary storage latency by four clock ticks. The scalar unit has six data-manipulation functional units: an integer/logical unit, a floating-point adder, a floating-point multiplier, a divide/square root functional unit, a front-end functional unit for multiplies and divides, and a binary/BCD converter. Scalar functional unit times appear to be very short, perhaps only three clock ticks. An instruction can be issued every clock tick.

The vector unit has two pipes, each of which contain an adder, a shifter, a logical unit, a divider, and a multiplier. In addition, one pipe has extra hardware to support operations like the sum of all elements, the product of all elements, the maximum of all elements, and the inner product of two vectors. Each pipe can be involved in one complex memory-to-memory transformation to a set of vectors at a time. For example, a pipe can read two vectors, multiply one by a scalar and subtract it from another, and then write the result back to memory. These complex transformations are implemented using several instructions, and data are "shortstopped" between functional units (the instructions are chained together, to use Cray's terminology). The use of "shortstopping" essentially inserts a delay line between functional units, as suggested in section 3.3. Vectors can be any length between one and 65536 elements. The vector processing times are quite long, due to the extensive setup, alignment, and routing among functional units that is performed. Consequently, vector processing is only efficient for long vectors.

Instructions are either 16, 32, or 64 bits in length, and most vector-instruction require 64 bits. Complex instructions, like the inner product of two vectors, are supported by the

hardware. In addition, the architecture has a number of data flag branch instructions that allow the program to branch to a section of code when an exceptional condition occurs, without the overhead of explicitly testing the condition. These instructions can begin execution a variable number of ticks after the event occurs, so they behave like the imprecise interrupts of the IBM System/360 Model 91.³ A vector control stream behaves like the vector mask register on the Cray architectures, except that, besides inhibiting the write of a masked-out result, it also prevents any flags from being set by that result's possible exceptional result.

6.6.2. Analysis

The ETA-10 represents a substantially different approach to supercomputing, than any of the other machines described, with its complex instruction set and combination of memory-to-memory vector instructions and register-to-register scalar instructions. Nevertheless, the analyses of this dissertation can still be applied.

The ETA-10 vector unit has a peak start-limit of ten, which can be achieved by starting two loads, one store, and two arithmetic operations in each of the two pipes. The scalar unit has a peak start-limit of one, because one instruction (specifying one operation) can be started each clock tick. Actual start-limits of eight or ten can be achieved for the vectorizable Livermore Kernels, as shown in table 6.9. The ETA-10 allows four load operations to be started each tick, as well as two store operations each tick. Each pipe can have up to two arithmetic operations starting each tick, so, for many of the Livermore Kernels, the peak start-limit can be achieved. With 256 scalar registers, the ETA-10 has enough temporary storage for most serial applications.

Table 6.9: Operation Types and Actual Start-Limits for Vectorizable Livermore Kernels Executing on the ETA-10.

Kernel	Operations	Actual Start-Limit
1	load, *, +, store	10
2	load, *, +, store	10
3	load, *, +	8
4	load, *, +	8
7	load, *, +, store	10
8	load, *, +, -, store	10
9	load, *, +, store	10
10	load, -, store	8
12	load, -, store	8

6.6.3. Execution Rates

The simulated execution rates for Livermore Kernels 1-14 on the ETA-10 are summarized in table 6.10.

Table 6.10: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the ETA-10.

Kernel	F.P. ops	Ticks	MFLOPS	X-MP MFLOPS
1	2000	700	408	191
2	360	220	234	183
3	2048	812	360	216
4	768	432	254	199
5	1992	5993	47.5	18.1
6	1994	5999	47.5	18.1
7	1920	640	429	204
8	1440	501	411	196
9	1700	621	391	185
10	900	560	230	104
11	999	3014	47.4	19.6
12	1000	580	246	115
13	1152	2485	66.2	47.1
14	1800	3341	77.0	49.8

The execution rates for vectorized loops is roughly half that of the NEC SX-2, because the ETA-10 has approximately the same speed clock but half the peak and actual start-limits. For scalar loops the speed is 50% to three times faster than the NEC SX-2, because of the shorter scalar execution pipelines.

The overall throughput of the eight-processor ETA-10 is over three gigaflops (GFLOPS) for the vectorized kernels. This is possible because each CPU has its own private high bandwidth memory that receives no interference from other CPUs.

CHAPTER 7

Architectural Integration

As an example of how close/distant register accessing, fast/bulk vector registers, and pipelined RAM (PRAM) chips can be integrated into a supercomputer architecture, the Cray-2 architecture is modified to incorporate 32 integrated vector/scalar registers with 4096 elements each. Data registers are organized in a close/distant hierarchy, so that some of them can be accessed in one clock tick while the rest can be read in two clock ticks and written in one clock tick.

The Cray-2, with its fast clock and long pipelines, is oriented towards fast vector execution, rather than towards fast scalar execution. The architectural modifications are intended to further increase the machine's vector performance, while neglecting scalar performance. In addition to the integrated vector/scalar registers, the add functional unit and the multiply functional unit are replicated so that two adds and two multiplies can be started each clock tick. Smaller, faster memory is used to support the bandwidth requirements of the four processors.

7.1. Cray-2 Architecture

The Cray-2²³ has four background processors and one foreground processor. These, plus a large memory, are immersed in liquid fluorocarbon in a small enclosure. Synchronization among processors is supported by 8 shared semaphore bits.

All functions are pipelined with a clock period of 4.1 nanoseconds, and pipelines are perhaps twice as long as those in the Cray-1 or the Cray X-MP. The functional units include an address adder, an address multiplier, a scalar shifter, a scalar logical unit, a vector logical

unit, a vector floating point adder, a scalar integer unit, a vector integer unit, and a floating point multiply/divide unit.

Each of the eight vector registers have 64 elements of 64-bit words, but they support no chaining at all. There are also eight scalar registers and eight 32-bit address registers. Sixteen kilowords of local memory with a four-tick access time can be used to store temporary results.

The Cray-2 has 256 million words of main memory shared among the five processors and organized in 128 interleaved banks. Memory has a 57-tick busy time and a 59-tick scalar load time. Each processor has only one port to main memory.

The Cray-2 is a load-store architecture with three-address instructions. Cray-2 instructions are specified using one, two, three, or five 16-bit parcels. The first parcel is used to specify the opcode and the operand registers, and any additional parcels are used to specify 16-bit, 32-bit, and 64-bit constant data or addresses. The opcode is generally seven bits long, although some flag control instructions and some single-operand instructions use the three least significant bits as an additional part of the opcode. Figure 7.1 shows the four instruction formats of the Cray-2, with the field sizes shown above them.

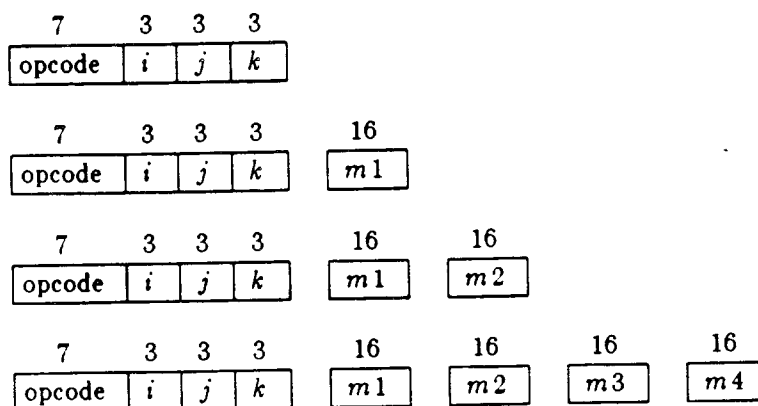


Figure 7.1: Instruction Formats of the Cray-2.

Each of the i , j , and k fields select one of eight registers of the type determined by the

opcode. In addition, k can specify part of the opcode, as discussed above. The i field usually specifies the destination of the instruction, and the j and k fields usually specify the source operands; however, for transfers of S registers or V registers to and from local or common memory, the i field always specifies the S register or V register and the j and k fields specify the address registers. For transfers of A registers to and from memory, the i field always specifies the destination of the transfer, and the k field always specifies the source of the transfer: a register or a memory address.

The complete instruction set of the Cray-2 is shown in table 7.2 at the end of the chapter. The entry x in an i , j , or k field indicates that those three bits are don't-cares.

7.2. LARC (Large Asymmetrical-Register Computer) Architecture

The Larc retains most of the architectural features of the Cray-2, except that the scalar and vector registers are replaced with more integrated registers, the scalar functional units are replaced with more vector functional units, and the large low-bandwidth memory is replaced with a smaller memory that can support three ports per processor.

7.2.1. Hardware Description

Each CPU has an address add and an address multiply functional unit, two integer functional units, two floating-point add functional units, and two multiply/reciprocal functional units. Except for the address add and multiply functional units, there are no functional units reserved for scalar operations. Each functional unit is the same as in the Cray-2, as described in Chapter 6. The replicated functional units are independent, and can be used for independent instructions at the same time. This is in contrast to the single-instruction parallel-execution (SIPE) functional units of the NEC SX-2,^{40,49,106} which must be used together for the same instruction. The replicated functional units are included because the same operations are present in several different partitions in many of the parallel Livermore Kernels⁶⁷

With replicated functional units, longer chains of vector operations can execute without interruptions due to busy functional units.

The Larc has 32 integrated full-width scalar/vector registers (F registers) of 4096 elements plus eight 32-bit scalar address registers (A registers). A vector-length register is associated with each F register, in addition to a global vector-length register for each CPU, and the vector mask register is 4096 bits long to match the maximum vector length. The F registers have enough bandwidth to allow chaining together vector instructions with any interval between instructions. Access to the first eight F registers and to all A registers can be made in one clock tick, while read access to the remaining F registers takes two clock ticks.

The longest vector length for the parallel kernels analyzed in Chapter 3 is 1024, and as many as 25 separate vector registers can be required. Serial kernels can require as many as 16 scalar registers.

These requirements are met by the set of 32 integrated F registers, each with 4096 elements. When the vector length for an F register is set to one, its first fast register is used as a scalar register, and when its vector length is greater than one, it behaves as a vector register. Access to the first element of each vector register is made shorter by using an unbalanced multiplexer with a fast path for new data to the output latch of the vector register. As described in Chapter 5, the fast registers and PRAM storage allow full chaining of vector instructions, with no restrictions on the chain-slot interval.

A pre-read instruction also allows starting vector reads at other than element zero. This is useful for applications like kernel 7, which operates on seven offset copies of the same vector. After a pre-read operation, the read counter is set to the pre-read address and is not reset at the beginning of a vector instruction's execution, as it normally is.

The vector mask (VM) register is 4096 bits long, corresponding to the maximum vector length of 4096 elements. It can be written to or read from an F register, using up to the first

64 elements. Writing a new VM can be overlapped with reading or using an old VM, to allow vector masks to be used while they are being updated.

With 32 vector registers and with replicated functional units, it is possible that several vector operations with different vector lengths can be executing at the same time. For this reason, and because integrated registers are also used as scalar registers, each integrated register has a vector length associated with it. In addition, there is a global vector length register (GVL) for the entire CPU.

When an F register is the destination of a scalar operation (such as copying an A register to an F register), the F register's vector length is set to one. If an F register is the destination of a vector instruction without an F-register source (such as a vector load), its vector length is set to (GVL). If an F register is the destination of a vector operation with one F-register source (such as reciprocal-square-root approximation), the vector length of the destination F register is set to the source vector length. If an F register is the destination of a vector operation with two F-register sources (such as adding the elements of two vectors together), the destination's vector length is set to that of the smaller of the F-register operands, unless one of the operands has length one, in which case the vector length is set to that of the larger of the two sources. This special case allows all elements of a vector to be operated upon by a scalar, supporting vector scaling and other operations. In addition, the vector length of any F register can be explicitly set to any value between one and 4096. The VM register has no vector length associated with it; in use, its length is determined by the length of the shorter non-scalar F-register operand if there is one or by (GVL) if there is no F-register operand.

F registers 0-7 and all A registers can be accessed in one clock tick, while F registers 8-31 require two ticks to read. With just eight of each type of register to be accessed in one clock tick, a short clock cycle is possible, and the analyses in section 3.4.2 indicate that hav-

ing some registers with two-tick access times does not significantly slow the execution of programs.

7.2.2. Larc Instruction Formats

The eight A registers are addressed normally with three-bit fields. However, with 16-bit parcels and a seven-bit opcode, there is not enough room for three five-bit fields to directly address the 32 F registers. The alternatives are to increase the length of instructions, or to address some F registers using short addresses and to address the others with longer addresses.

The NEC SX-2 uses a 32-bit instruction format for its short instructions; this allows its eight-bit register addresses to be specified directly. This also requires twice the instruction stream bandwidth, as does a 16-bit instruction format, and an instruction buffer twice the size for the same performance. Increasing the basic instruction size from 16 bits to 22 bits to accommodate five-bit register addresses would not require excessive instruction stream bandwidth, but instructions would not necessarily align on 64-bit-word boundaries, complicating branches and instruction fetching. A separate instruction memory could have any word size desired, but this would depart significantly from the original architecture.

It would be possible to use a 16-bit instruction format when only registers 0-8 are addressed, and to use a 32-bit instruction format if any of the registers specified include the other 24 registers. However, this treats the registers non-uniformly and requires two formats for each instruction. Another way to address the 32 F registers is to address one F register explicitly using a five-bit specifier, and specify the remaining register or registers indirectly, using fewer bits. This method also requires two formats for most instructions, but it allows all registers to be addressed using a 16-bit instruction format. This method is chosen for the Larc architecture.

The F register destination is explicitly specified using five bits. Each time an instruction with an F register destination issues, its address is added to a queue of F register addresses. F-register source specifiers are two bits long and they select which of the last four F registers written is to be used as the operand. If the desired F register is not one of the last four written, a two-parcel instruction must be used. The sequence of F registers written can be determined at compile time, so the correct specification can be determined then, as well. In addition, each F-register destination can be determined before the instruction actually issues, so the queue can be updated enough in advance that instructions can be issued without delays. Furthermore, results tend to be used soon after they are written, so very often the desired result is one of the last four written. This indirect addressing mechanism is also used when A registers are specified in the same instruction as F registers.

The instruction formats of the Larc are shown in figure 7.2.

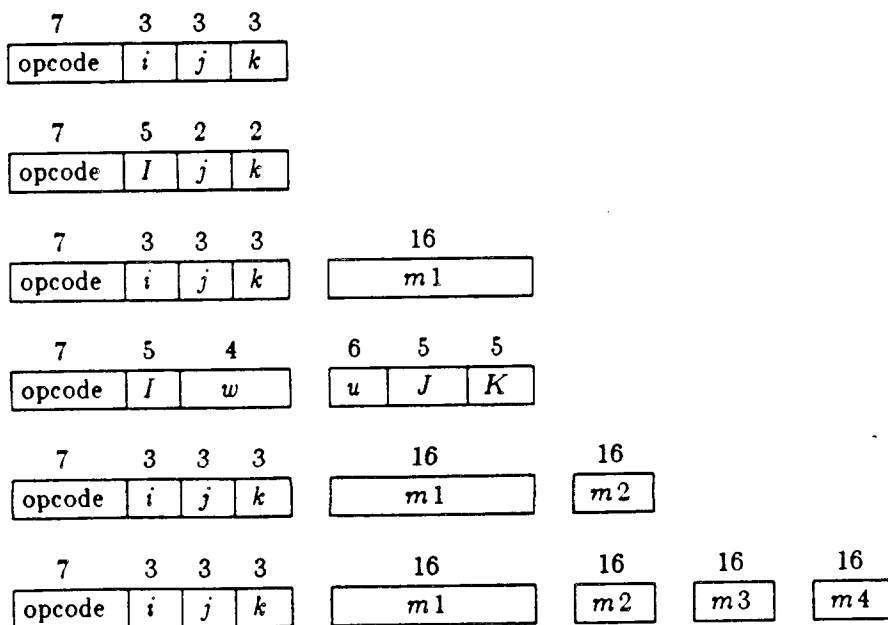


Figure 7.2: Instruction Formats of the Larc.

The *i* field has three bits, as in the Cray-2 architecture, but the *j* and *k* fields have either three bits or two bits, depending on the instruction format. If the first field is a three-bit *i*

field, the j and k fields also have three bits, but if the first field is a five-bit I field, the j and k fields have two bits each. The J and K fields in the two-parcel instruction format have five bits each. Don't-care fields of four bits are indicated by w , and don't-care fields of six bits are indicated by u .

The instruction formats of the Larc are slightly more complex than those of the Cray-2; however, for a given type of register specification, the address specification comes from at most two places. The instruction format can be determined from the first seven bits of the opcode, so elaborate decoding hardware is unnecessary for fast instruction issuing. Furthermore, address specification decoding time can be pipelined into the instruction fetch process so instructions can be issued at a fast rate. Branch latency may be higher with this more complex address specification mechanism, due to a potentially longer instruction issue pipeline, but branches are relatively infrequent in applications for which vector execution is appropriate.

The complete instruction set of the Larc is shown in table 7.3 at the end of the chapter. Six-bit, five-bit, four-bit, three-bit, two-bit, and one-bit don't-care fields are indicated by u , v , w , x , y , and z , respectively. The opcodes saved by eliminating explicit scalar-register instructions and scalar-vector-register instructions are used to specify one-parcel and two-parcel F-register instructions.

The pre-read instructions discussed above use opcodes 104 and 105; and the register-specific vector-length instructions use opcodes 106, 107, 110, and 111. These allow vector lengths to be set to either constant values or to the contents of an A register. In addition, vector-element-access instructions have been added, using opcodes 130, 131, 132, and 133. These allow transfers of particular vector elements to and from the first element of an F register.

Except for these additions, the instruction set of the Larc is functionally identical to that of the Cray-2. Thus, more registers can be added to an architecture without a significant impact on the instruction set.

7.3. Discussion

By combining scalar and vector registers into integrated registers, a larger number of registers of each type are made available. During serial execution up to 32 scalar registers are available for temporary results, and during parallel execution up to 32 vector registers can be used. The large number of registers allows computations to execute without running out of temporary storage, saving time and memory bandwidth that would otherwise be spent saving and restoring temporary results. They also allow more instructions to execute overlapped. The long vector register length made possible by their fast/bulk organization allows most vectors to be processed by a single sequence of vector instructions, reducing both instruction stream bandwidth requirements and pipeline start-up costs.

The unrestricted chaining made possible by the high-bandwidth integrated registers allows multiple operations on vectors to be overlapped. This, coupled with the replicated independent functional units and the higher memory bandwidth, allows a high start-limit for many programs, that results in faster execution.

The higher memory-bandwidth is achieved by using memory banks with shorter busy times. This requires higher-power RAM chips, less-dense RAM chips, or PRAM chips. In all these cases, the main memory size is smaller than it would be if the highest-density RAM chips were used: perhaps one fourth the size. However, a smaller, high-bandwidth memory system can be supplemented with a large, high-bandwidth backing store. During transfers to and from the backing store, the memory bandwidth available to the CPUs may be reduced to that of a larger direct-access memory, but most of the time the full memory bandwidth is available for computations. This approach affects programming convenience for applications

that require random access to data sets larger than the main memory size. However, an application requiring more than n memory locations is also likely to require more than $4n$ memory locations, so the inconvenience of a smaller memory is negligible.

The replicated functional units would require extra hardware that would require more space and power. Some of this could be recovered from the scalar functional units eliminated from the CPU; the net effect to increase the amount of hardware by approximately 10%. Additional space and cooling area could be obtained by stretching out the functional units radially, as described in Chapter 4. This would allow the entrances and exits of pipelines to be physically adjacent to the center of the CPUs, while most of their hardware is towards the periphery where there is more space. The longer pipelines used in the Cray-2 and in the Larc lend themselves to this radial stretching.

7.4. Execution Rates

Simulated execution rates of the Livermore Kernels executing on the Larc are shown in Table 7.1. The same simulator that was used in Chapter 6 is used here. Execution rates for serial kernels 5, 6, and 11 are the same as on the Cray-2, because the same long arithmetic pipelines are used for both machines. Execution rates for kernels 13 and 14 are almost twice those of the Cray-2, because main memory with lower latency is used in the Larc (execution rates for kernels 13 and 14 using local memory for array storage would be identical to those of the Cray-2 using local memory). Execution rates of all other kernels are a factor of 1.5-3 times higher for the Larc than the Cray2, because memory has adequate bandwidth and half the latency and because there are two multipliers and two adders in the Larc, which increase its actual start-limit for the parallel applications.

Table 7.1: Simulated Execution Rates for Livermore Kernels 1-14 Executing on the Larc.

Kernel	Floating Point Ops	Ticks	MFLOPS	Cray-2 MFLOPS	Ratio
1	2000	868	549	367	1.49
2	360	268	320	172	1.86
3	2048	1272	383	210	1.82
4	768	578	316	185	1.70
5	1992	35889	13.2	13.2	1.00
6	1994	35925	13.2	13.2	1.00
7	1920	668	684	388	1.76
8	1440	548	626	362	1.72
9	1700	716	565	312	1.81
10	900	1032	208	107	1.94
11	999	18015	13.2	13.2	1.00
12	1000	1050	269	119	2.26
13	1152	6545	41.9	26.5	1.58
14	1800	10317	41.5	29.1	1.42

The system throughput of a four-processor Larc is four times that of one processor, because memory bandwidth is adequate to handle all four processor's simultaneous memory accesses. The system throughput of the Larc is more than twice that of the NEC SX-2 and almost equal to that of an eight-processor ETA-10; the Larc exceeds 2.7 GFLOPS for some applications.

Table 7.2: Cray-2 Instruction Set.

Opcode	Extension	Mnemonic	Description
000xjk		err	Error exit
001xjk		exit	Normal exit
002irk		r, Ai Ak	Register jump to (Ak) with return address in Ai
003xxx	m1 m2	j exp	Unconditional jump to m1 m2
004xxx	m1 m2	jcs exp	Jump to m1 m2 if Semaphore clear, set Semaphore
005xxx	m1 m2	jss exp	Jump to m1 m2 if Semaphore set, set Semaphore
006xxx		ssm	Set Semaphore
007xxx		csm	Clear Semaphore
010xxk	m1 m2	jz Ak, exp	Branch if (Ak) is zero
011xxk	m1 m2	jn Ak, exp	Branch if (Ak) is nonzero
012xxk	m1 m2	jp Ak, exp	Branch if (Ak) is positive
013xxk	m1 m2	jm Ak, exp	Branch if (Ak) is negative
014xjz	m1 m2	jz Sj, exp	Branch if (Sj) is zero
015xjz	m1 m2	jn Sj, exp	Branch if (Sj) is nonzero
016xjz	m1 m2	jp Sj, exp	Branch if (Sj) is positive
017xjz	m1 m2	jm Sj, exp	Branch if (Sj) is negative
020ijk		Ai Aj+Ak	Integer sum of (Aj) and (Ak) to Ai
021ijk		Ai Aj-Ak	Integer difference of (Aj) and (Ak) to Ai
022ijk		Ai Aj*Ak	Integer product of (Aj) and (Ak) to Ai
023ijk		Ai Aj*Ak	Integer product of (Aj) and (Ak) to Ai
024ijz		Ai Sj	Copy (Sj) to Ai
025izz		Ai VL	Copy (VL) to Ai
026ijk		Ai exp	Load Ai with 6-bit value
027ijk		Ai exp	Load Ai with 6-bit negative value
030xxk		VM Vk, z	Set VM from zero elements of (Vk)
031xxk		VM Vk, n	Set VM from nonzero elements of (Vk)
032xxk		VM Vk, p	Set VM from positive elements of (Vk)
033xxk		VM Vk, m	Set VM from negative elements of (Vk)
034xjz		VM Sj	Copy (Sj) to VM
035xx0		dri	Disable halt on memory field range error
035xx1		eri	Enable halt on memory field range error
035xx2		dfi	Disable halt on floating-point error
035xx3		efi	Enable halt on floating-point error
036xxk		VL Ak	Copy (Ak) to VL
037xxk		VL Ak	Copy (Ak) to VL
040izz	m1	Ai exp	Load Ai with 16-bit positive value
041izz	m1	Ai exp	Load Ai with 16-bit negative value
042izz	m1 m2	Ai exp	Load Ai with 32-bit value
043izz	m1 m2	Ai exp	Load Ai with 32-bit value
044izz	m1	Ai [exp]	Read from location exp in Local Mem. to Ai
045xxk	m1	[exp] Ak	Write (Ak) to location exp in Local Mem.
046irk		Ai [Ak]	Read from location (Ak) in Local Mem. to Ai
047xjk		Ak [Aj]	Write (Aj) to location Ak in Local Mem.

Table 7.2: Cray-2 Instruction Set (Continued).

Opcode	Extension	Mnemonic	Description
050izz	m1 m2	Si exp	Load Si with a 32-bit positive value
051izz	m1 m2	Si exp	Load Si with a 32-bit negative value
052izz	m1 m2	Si exp	Load Si left side with a 32-bit value
053izz	m1 m2 m3 m4	Si exp	Load Si with a 64-bit value
054izz	m1	Si [exp]	Read from location exp in Local Mem.
055zjz	m1	[exp] Sj	Write (Sj) to location exp in Local Mem.
056izk		Si [Ak]	Read from location (Ak) in Local Mem.
057izk		[Ak] Si	Write (Si) to location (Ak) in Local Mem.
060ijk		Si (Aj, Ak)	Read from Common Mem. at loc. (Aj) + (Ak) to Si
061ijk		(Aj, Ak) Si	Write (Si) to Common Mem. at loc. (Aj) + (Ak)
062izk		Si (Ak)	Read from Common Mem. at location (Ak) to Si
063izk		(Ak) Si	Write (Si) to Common Mem. at location (Ak)
064izk	m1 m2	Si (Ak, exp)	Read from Common Mem. location (Ak)+exp to Si
065izk	m1 m2	(Ak, exp) Si	Write (Si) to Common Mem. at location (Ak)+exp
066izz	m1 m2	Si (exp)	Read from Common Mem. location exp to Si
067izz	m1 m2	(exp) Si	Write (Si) to Common Mem. at location exp
070ijk		Vi (Aj, Ak)	Read from Common Mem. loc. (Aj) with stride (Ak) to Vi
071ijk		(Aj, Ak) Vi	Write (Vi) to Common Mem. loc. (Aj) with stride (Ak)
072ijk		Vi (Ak, Vj)	Gather from Common Mem. locations (Ak)+(Vj) to Vi
073ijk		(Ak, Vj) Vi	Scatter (Vi) to Common Mem. locations (Ak)+(Vj)
074izk		Vi [Ak]	Read from Local Mem. location (Ak) to Vi
075izk		[Ak] Vi	Write (Vi) to Local Mem. location (Ak)
076ijk		pass exp	Pass
077ijk		pass exp	Pass
100ijk		Si Sj&Sk	Logical product of (Sj) and (Sk) to Si
101ijk		Si #Sk&Sj	Logical product of (Sj) and complement (Sk) to Si
102ijk		Sj*Sk	Logical difference of (Sj) and (Sk) to Si
103ijk		Sj!Sk	Logical sum of (Sj) and (Sk) to Si
104ijk		Si Sj+Sk	Integer sum of (Sj)+(Sk) to Si
105ijk		Si Sj-Sk	Integer difference (Sj)-(Sk) to Si
106ij0		Si pSj	Population count of (Sj) to Si
106ij1		Si qSj	Population count parity of (Sj) to Si
107ijz		Si zSj	Leading zero count of (Sj) to Si
110ijk		Si Si<exp	Shift (Si) left exp=64-jk places to Si
111ijk		Si Si>exp	Shift (Si) right exp=jk places to Si
112ijk		Si Si,Sj<Ak	Shift (Si and Sj) left (Ak) places to Si
113ijk		Sj,Si>Ak	Shift (Si and Sj) right (Ak) places to Si
114izz		Si VM	Transmit (VM) to Si
115izz		Si rt	Transmit real-time clock to Si
116ijk		Si exp	Load Si with 6-bit positive value
117ijk		Si exp	Load Si with 6-bit negative value
120ijk		Si Sj+fSk	Floating-point sum of (Sj) and (Sk) to Si
121ijk		Si Sj-fSk	Floating-point difference (Sj)-(Sk) to Si
122izk		Si fx,Sk	Convert (Sk) from floating-point to integer to Si
123izk		Si ft,Sk	Convert (Sk) from integer to floating-point to Si

Table 7.2: Cray-2 Instruction Set (Continued).

Opcode	Extension	Mnemonic	Description
124ijk		$S_i S_j * f S_k$	Floating-point product of (S_j) and (S_k) to S_i
125ijk		$S_i S_j * f S_k$	Floating-point product of (S_j) and (S_k) to S_i
126ijk		$S_i S_j * i S_k$	Reciprocal iteration of $2-(S_j)*(S_k)$ to S_i
127ijk		$S_i S_j * q S_k$	Reciprocal square root iteration $[3-(S_j)*(S_k)]/2$ to S_i
130izk		$S_i A_k$	Transmit (A_k) to S_i with no Sign extension
131izk		$S_i + A_k$	Transmit (A_k) to S_i with Sign extension
132ijz		$S_i / h S_j$	Floating-point reciprocal approx. of (S_j) to S_i
133ijz		$S_i * q S_j$	Floating-point reciprocal square root approx. of (S_j) to S_i
134zzz		pass	Pass
135zzz		pass	Pass
136zzz		pass	Pass
137zzz		pass	Pass
140ijk		$V_i S_j \& V_k$	Logical products of (S_j) and (V_k) to V_i
141ijk		$V_i V_j \& V_k$	Logical products of (V_j) and (V_k) to V_i
142ijk		$V_i S_j * V_k$	Logical differences of (S_j) and (V_k) to V_i
143ijk		$V_i V_j * V_k$	Logical differences of (V_j) and (V_k) to V_i
144ijk		$V_i S_j ! V_k$	Logical sums of (S_j) and (V_k) to V_i
145ijk		$V_i V_j ! V_k$	Logical sums of (V_j) and (V_k) to V_i
146ijk		$V_i S_j ! V_k \& VM$	Transmit (S_j) if VM bit=1; (V_k) if VM bit=0 to V_i
147ijk		$V_i V_j ! V_k \& VM$	Transmit (V_j) if VM bit=1; (V_k) if VM bit=0 to V_i
150ijk		$V_i V_j < A_k$	Shift (V_j) left (A_k) bits with zero fill to V_i
151ijk		$V_i V_j > A_k$	Shift (V_j) right (A_k) bits with zero fill to V_i
152ijk		$V_i V_j, V_j < A_k$	Double shift (V_j) left (A_k) places to V_i
153ijk		$V_i V_j, V_j > A_k$	Double shift (V_j) right (A_k) places to V_i
154ijk		$V_i S_j * f V_k$	Floating-point products of (S_j) and (V_k) to V_i
155ijk		$V_i V_j * f V_k$	Floating-point products of (V_j) and (V_k) to V_i
156ijk		$V_i V_j * i V_k$	Reciprocal iteration of $2-(V_j)*(V_k)$ to V_i
157ijk		$V_i V_j * q V_k$	Reciprocal square root iteration $[3-(V_j)*(V_k)]/2$ to V_i
160ijk		$V_i S_j + V_k$	Integer sums of (S_j) and (V_k) to V_i
161ijk		$V_i V_j + V_k$	Integer sums of (V_j) and (V_k) to V_i
162ijk		$V_i S_j - V_k$	Integer differences of (S_j) and (V_k) to V_i
163ijk		$V_i V_j - V_k$	Integer differences of (V_j) and (V_k) to V_i
164ij0		$V_i p V_j$	Population counts of (V_j) to V_i
164ij1		$V_i q V_j$	Population count parity of (V_j) to V_i
165ijz		$V_i z V_j$	Leading zero count of (V_j) to V_i
166izk		$V_i / h V_k$	Floating-point reciprocal approx. of (V_k) to V_i
167izk		$V_i * q V_k$	Floating-point reciprocal square root approx. of (V_k) to V_i
170ijk		$V_i S_j + f V_k$	Floating-point sum of (S_j) and (V_k) to V_i
171ijk		$V_i V_j + f V_k$	Floating-point sum of (V_j) and (V_k) to V_i
172ijk		$V_i S_j - f V_k$	Floating-point differences of (S_j) and (V_k) to V_i
173ijk		$V_i V_j - f V_k$	Floating-point differences of (V_j) and (V_k) to V_i
174izk		$V_i f x, V_k$	Integer form of floating-point (V_k) to V_i
175izk		$V_i f t, V_k$	Floating-point form of integer (V_k) to V_i
176ijk		$V_i c_i, S_j \& S_k$	Enter V_i with compressed iota S_j and S_k
177ijk		$V_i c_i, S_j \& S_k$	Enter V_i with compressed iota S_j and S_k

Table 7.3: Larc Instruction Set.

Opcode	Extension	Mnemonic	Description
000zjk		err	Error exit
001zjk		exit	Normal exit
002izk		r, Ai Ak	Register jump to (Ak) with return address in Ai
003zzz	m1 m2	j exp	Unconditional jump to m1 m2
004zzz	m1 m2	jcs exp	Jump to m1 m2 if Semaphore clear, set Semaphore
005zzz	m1 m2	jss exp	Jump to m1 m2 if Semaphore set, set Semaphore
006zzz		ssm	Set Semaphore
007zzz		csm	Clear Semaphore
010zzk	m1 m2	jz Ak, exp	Branch if (Ak) is zero
011zzk	m1 m2	jn Ak, exp	Branch if (Ak) is nonzero
012zzk	m1 m2	jp Ak, exp	Branch if (Ak) is positive
013zzk	m1 m2	jm Ak, exp	Branch if (Ak) is negative
014lw	m1 m2	jz Fi0, exp	Branch if (Fi0) is zero
015lw	m1 m2	jn Fi0, exp	Branch if (Fi0) is nonzero
016lw	m1 m2	jp Fi0, exp	Branch if (Fi0) is positive
017lw	m1 m2	jm Fi0, exp	Branch if (Fi0) is negative
020ijk		Ai Aj+Ak	Integer sum of (Aj) and (Ak) to Ai
021ijk		Ai Aj-Ak	Integer difference of (Aj) and (Ak) to Ai
022ijk		Ai Aj*Ak	Integer product of (Aj) and (Ak) to Ai
023ijk		Ai Aj/Ak	Integer product of (Aj) and (Ak) to Ai
024lzk		Ak Fi0	Copy (Fi0) to Ak
025izz		Ai GVL	Copy (GVL) to Ai
026ijk		Ai exp	Load Ai with 6-bit value
027ijk		Ai exp	Load Ai with 6-bit negative value
030lw		VM Fi, z	Set VM from zero elements of (Fi)
031lw		VM Fi, n	Set VM from nonzero elements of (Fi)
032lw		VM Fi, p	Set VM from positive elements of (Fi)
033lw		VM Fi, m	Set VM from negative elements of (Fi)
034lw		VM Fi0	Copy (Fi0) to VM
035zz0		dri	Disable halt on memory field range error
035zz1		eri	Enable halt on memory field range error
035zz2		dfi	Disable halt on floating-point error
035zz3		efi	Enable halt on floating-point error
036zzk		GVL Ak	Copy (Ak) to GVL
037zzk		GVL Ak	Copy (Ak) to GVL
040izz	m1	Ai exp	Load Ai with 16-bit positive value
041izz	m1	Ai exp	Load Ai with 16-bit negative value
042izz	m1 m2	Ai exp	Load Ai with 32-bit value
043izz	m1 m2	Ai exp	Load Ai with 32-bit value
044izz	m1	Ai [exp]	Read from location exp in Local Mem. to Ai
045zzk	m1	[exp] Ai	Write (Ak) to location exp in Local Mem.
046izk		Ai [Ak]	Read from location Ak in Local Mem. to Ai
047zjk		Ak [Aj]	Write (Aj) to location Ak in Local Mem.

Table 7.3: Larc Instruction Set (Continued).

Opcode	Extension				Mnemonic	Description
050lw	m1	m2			Fi0 exp	Load Fi0 with a 32-bit positive value
051lw	m1	m2			Fi0 exp	Load Fi0 with a 32-bit negative value
052lw	m1	m2			Fi0 exp	Load Fi0 left side with a 32-bit value
053lw	m1	m2	m3	m4	Fi0 exp	Load Fi0 with a 64-bit value
070ljk					Fi (Aj, Ak)	Read from Common Mem. loc. (Aj), stride (Ak) to Fi
071lw	uJK				Fi (Aj, Ak)	Read from Common Mem. loc. (Aj), stride (Ak) to Fi
072ljk					(Aj, Ak) Fi	Write (Fi) to Common Mem. loc. (Aj) with stride (Ak)
073lw	uJK				(Aj, Ak) Fi	Write (Fi) to Common Mem. loc. (Aj) with stride (Ak)
074ljk					Fi (Ak, Fj)	Gather from Common Mem. locations (Ak)+(Fj) to Fi
075lw	uJK				Fi (Ak, Fj)	Gather from Common Mem. locations (Ak)+(Fj) to Fi
076ljk					(Ak, Fj) Fi	Scatter (Fi) to Common Mem. locations (Ak)+(Fj)
077lw	uJK				(Ak, Fj) Fi	Scatter (Fi) to Common Mem. locations (Ak)+(Fj)
100lzk					Fi [Ak]	Read from Local Mem. location (Ak) to Fi
101lzk					[Ak] Fi	Write (Fi) to Local Mem. location (Ak)
102ijk					pass exp	Pass
103ijk					pass exp	Pass
104ljk					pr Fi, exp	Pre-read first exp elements of Fi
105lzx	m1				pr Fi, exp	Pre-read first exp elements of Fi
106lzk					VLi Ak	Set Vector Length of Fi to (Ak)
107lzk					AK VLi	Set AK to Vector Length of Fi
110ljk					VLi jk	Set Vector Length of Fi to exp
111lw	m1				VLi jk	Set Vector Length of Fi to exp
112lzk					Fi0 Ak	Transmit (Ak0) to Fi with no Sign extension
113lzk					Fi0 +Ak	Transmit (Ak0) to Fi with Sign extension
114lw					Fi VM	Transmit (VM) to Fi
115lw					Fi0 rt	Transmit real-time clock to Fi0
116ljk					Fi0 exp	Load Fi0 with 4-bit positive value
117ljk					Fi0 exp	Load Fi0 with 4-bit negative value
120ljk					Fi Fj&Fk	Logical products of (Fj) and (Fk) to Fi
121lw	uJK				Fi Fj&Fk	Logical products of (Fj) and (Fk) to Fi
122ljk					Fi Fj*Fk	Logical differences of (Fj) and (Fk) to Fi
123lw	uJK				Fi Fj*Fk	Logical differences of (Fj) and (Fk) to Fi
124ljk					Fi Fj!Fk	Logical sums of (Fj) and (Fk) to Fi
125lw	uJK				Fi Fj!Fk	Logical sums of (Fj) and (Fk) to Fi
126ljk					Fi Fj!Fk&VM	Transmit (Fj) if VM bit=1; (Fk) if VM bit=0 to Fi
127lw	uJK				Fi Fj!Fk&VM	Transmit (Fj) if VM bit=1; (Fk) if VM bit=0 to Fi

Table 7.3: Larc Instruction Set (Continued).

Opcode	Extension	Mnemonic	Description
130ljk		Fi,Ak Fj0	Transmit Fj0 to element (Ak) of Fi
131lw	uJK	Fi,Ak Fj0	Transmit Fj0 to element (Ak) of Fi
132ljk		Fi0 Fj, Ak	Transmit element (Ak) of Fj to Fi0
133lw	uJK	Fi0 Fj, Ak	Transmit element (Ak) of Fj to Fi0
134ljk		Fi Fj < Ak	Shift (Fj) left (Ak) bits with zero fill to Fi
135lw	uJK	Fi Fj < Ak	Shift (Fj) left (Ak) bits with zero fill to Fi
136ljk		Fi Fj > Ak	Shift (Fj) right (Ak) bits with zero fill to Fi
137lw	uJK	Fi Fj > Ak	Shift (Fj) right (Ak) bits with zero fill to Fi
140ljk		Fi Fj, Fj < Ak	Double shift (Fj) left (Ak) places to Fi
141lw	uJK	Fi Fj, Fj < Ak	Double shift (Fj) left (Ak) places to Fi
142ljk		Fi Fj, Fj > Ak	Double shift (Fj) right (Ak) places to Fi
143lw	uJK	Fi Fj, Fj > Ak	Double shift (Fj) right (Ak) places to Fi
144ljk		Fi Fj * fFk	Floating-point products of (Fj) and (Fk) to Fi
145lw	uJK	Fi Fj * fFk	Floating-point products of (Fj) and (Fk) to Fi
146ljk		Fi Fj * iFk	Reciprocal iteration of $2 \cdot (Fj) \cdot (Fk)$ to Fi
147lw	uJK	Fi Fj * iFk	Reciprocal iteration of $2 \cdot (Fj) \cdot (Fk)$ to Fi
150ljk		Fi Fj * qFk	Reciprocal square root iteration $[3 \cdot (Fj) \cdot (Fk)] / 2$ to Fi
151lw	uJK	Fi Fj * qFk	Reciprocal square root iteration $[3 \cdot (Fj) \cdot (Fk)] / 2$ to Fi
152ljk		Fi Fj + Fk	Integer sums of (Fj) and (Fk) to Fi
153lw	uJK	Fi Fj + Fk	Integer sums of (Fj) and (Fk) to Fi
154ljk		Fi Fj - Fk	Integer differences of (Fj) and (Fk) to Fi
155lw	uJK	Fi Fj - Fk	Integer differences of (Fj) and (Fk) to Fi
156l0k		Fi pFk	Population counts of (Fk) to Fi
156l1k		Fi qFk	Population count parity of (Fk) to Fi
157l0y	uvK	Fi pFk	Population counts of (Fk) to Fi
157l1y	uvK	Fi qFk	Population count parity of (Fk) to Fi
160lyk		Fi zFk	Leading zero count of (Fk) to Fi
161lw	uvK	Fi zFk	Leading zero count of (Fk) to Fi
162lyk		Fi /hFk	Floating-point reciprocal approx. of (Fk) to Fi
163lw	uvK	Fi /hFk	Floating-point reciprocal approx. of (Fk) to Fi
164lyk		Fi *qFk	Floating-point reciprocal square root approx. of (Fk) to Fi
165lw	uvK	Fi *qFk	Floating-point reciprocal square root approx. of (Fk) to Fi
166ljk		Fi Fj + fFk	Floating-point sum of (Fj) and (Fk) to Fi
167lw	uJK	Fi Fj + fFk	Floating-point sum of (Fj) and (Fk) to Fi
170ljk		Fi Fj - fFk	Floating-point differences of (Fj) and (Fk) to Fi
171lw	uJK	Fi Fj - fFk	Floating-point differences of (Fj) and (Fk) to Fi
172lyk		Fi fx, Fk	Integer form of floating-point (Fk) to Fi
173lw	uvK	Fi fx, Fk	Integer form of floating-point (Fk) to Fi
174lyk		Fi ft, Fk	Floating-point form of integer (Fk) to Fi
175lw	uvK	Fi ft, Fk	Floating-point form of integer (Fk) to Fi
176ljk		Fi ci, Fj0 & Fk0	Enter Fi with compressed iota Fj0 and Fk0
177lw	uJK	Fi ci, Fj0 & Fk0	Enter Fi with compressed iota Fj0 and Fk0

CHAPTER 8

Concluding Remarks

8.1. Summary of Results

8.1.1. Summary of Program Characteristics

As discussed in section 3.3.3, the serial programs of the Livermore Kernels⁸⁷ need short operation execution times for fast execution, while the parallel programs need to start many operations at a time for fast execution. (Cf. theorem 3.6, theorem 3.7.) Short execution times and many starts at a time are not necessarily required at the same time.

Many memory accesses can be started well before their results are needed, so higher-latency, high-bandwidth memory can be used for them. This requires a large enough set of fast-access temporary storage to hold the results of memory reads before they are used. (Cf. section 3.3.4.) Several hundred temporary results can be maintained by some programs with some schedules, so several hundred temporary storage locations that are faster than main memory can be effectively used to speed up execution. (Cf. sections 3.3.5 and 3.4.1.) Most results are used immediately after they are generated, so much of the temporary storage can be optimized for speed and density without regard for long-term data integrity. (Cf. section 3.3.5.) Also, most results are only used once, so specialized or implicit result transmission can be used much of the time. (Cf. section 3.3.5.)

Most of the temporary storage can be slow, relative to fast registers. Most programs that use a few fast registers but mostly use registers that have two to four times the access time of fast registers can execute essentially as fast as programs that use all fast registers. (Cf. section 3.4.2.)

As discussed in section 3.4.2, computers with pure memory-to-memory architectures and no vector addressing mechanism have a maximum execution speed that is at most one third to two thirds the speed of computers with many programmer-addressable registers. This is because of the long access times for main memory and because of the instruction-stream bandwidth required to specify full memory addresses. The instruction stream and memory bandwidth problem can sometimes be eliminated by using complex vector instructions, such as $\bar{x} = \bar{y} - a * \bar{z}$, that do not require storing any temporary or intermediate results. However, there are far more complex instructions than there are basic instructions that perform the same function, so instruction sets must be larger. Furthermore, complex instructions do not save memory bandwidth when temporary results are re-used.

The hardware requirements and a close approximation of the execution times of programs can be predicted from the partitionings of the dependency graphs. Analyses of these condensed specifications of programs are both faster and more generally applicable than analyses and simulations of the full dependency graphs. (Cf. sections 3.3 and 3.4.)

8.1.2. Datapath Design

Physical signal propagation delays, logic propagation delays, selection delays, and pipeline latch overhead limit the speed at which computers can execute. The influence of the parameters that affect execution speed can be analyzed, and optimal choices of the parameters can be made for a given workload.

As discussed in section 4.1, propagation distances in a plane grow at a rate between the square root and linearly with the number of devices, and in three dimensions distances grow at a rate between the cube root and the square root of the number of devices. (Cf. equations 4.2 and 4.4.) With a large enough number of devices and random communication among them, propagation delays dominate the execution time. However, by clustering devices that communicate with each other and by stretching linear pipelines out radially, propagation

delays can be kept small.

As discussed in section 4.2, data selection and routing delays grow as the log of the number of devices, and, in practice, multiplexing delays dominate. By locating the subset of devices logically closer to the roots of the fan-in and fan-out trees, the delays to or from a subset of the devices can be made smaller than the delays to or from the other devices. This technique facilitates the implementation of fast and slow registers discussed above.

As discussed in section 4.3.2, if many data are processed at a time, pipelines should have many stages with few levels of logic in each; however, if few data are processed at a time, pipelines should have few stages, with many levels of logic in each. The optimal number of levels of logic in each pipeline stage grows as the square root of the number of gate delays to perform the function divided by the number of data processed at a time. (Cf. equation 4.18.) However, for typical gate characteristics, the shortest practical pipeline stage has approximately four levels of logic.

Pipelines with variable numbers of stages are not practical to implement, because the logic to switch the number of stages increases the pipeline latency more than switching out stages reduces it. (Cf. theorem 4.1.)

If several identical pipelines are used, the optimal number of pipeline stages increases by the square root of the number of identical pipelines. (Cf. equation 4.24.) Identical pipelines that are independent can be used effectively by more programs than identical pipelines that must be used together for the same instruction. (Cf. section 4.3.2.3.)

As discussed in section 4.3.3, if functional units are implemented on a single chip, the maximum chip-I/O rate may be much smaller than the maximum clock rate on the chip. Storing a small number of values on the functional unit chips reduces the I/O requirements for some applications. Nonlinear pipelines running at fast clock rates can offer more efficient use of the hardware than linear pipelines running at slower clock rates, and the reduced

initiation rate of nonlinear pipelines matches the reduced I/O rate well.

8.1.3. Temporary Storage Devices and Access Mechanisms

Registers can be organized in a close/distant hierarchy that allows some registers to be accessed quickly, while other registers are accessed more slowly. However, the bandwidth of the set of registers is the same as if they all had fast access times. This close/distant access structure allows an efficient implementation of the fast and slow registers discussed above. (Cf. section 5.1.)

Vector registers can be implemented using high density, relatively slow memory, as long as the first few elements are stored in fast registers or memory. (Cf. theorem 5.1.) These fast/bulk vector registers provide low-latency access to the first few elements, while providing high-bandwidth access to all elements. The advantages of fast/bulk vector registers over traditional vector registers are that they provide longer vector registers for the same access-latency and access-bandwidth, they support lower-latency, higher-bandwidth access for the same length vector registers, and they can require fewer chips to implement for the same length and for the same access-latency and access-bandwidth. (Cf. section 5.2.1.)

As discussed in section 5.2.2.2, fast/bulk vector registers can be generalized to provide multiple-port access so that they can support general chaining between vector instructions. All elements of the vector are stored in high-density, relatively-slow memory, and a few vector elements are also stored in fast registers. The fast registers are used to chain writes to reads.

Access to random-access-memory (RAM) can be pipelined, increasing the bandwidth of access by a factor of four or more and increasing the access-latency by 50%. (Cf. section 5.3.3.) Pipelined random-access-memory (PRAM) chips have approximately one half the density of non-pipelined RAM chips using the same technology. (Cf. section 5.3.3.) PRAM is better than interleaving because its access-bandwidth is independent of the history of access,

so no restrictions on access patterns are necessary. (Cf. theorem 5.3.) PRAM is particularly well-suited for vector storage or for memory systems that are traditionally implemented using interleaved memory banks. (Cf. section 5.4.)

8.1.4. Architectural Integration

The lengths of vector registers can be increased with virtually no changes to the architecture: only vector mask registers and provisions for setting the vector length must be changed. (Cf. section 7.2.)

Adding unrestricted chaining to the vector registers is invisible at the architectural level, but it increases the potential execution rate significantly. (Cf. section 7.4.) If functional units are replicated and a large number of vector registers are made available, provisions for simultaneous vector instruction execution with several different vector lengths are useful. (Cf. section 7.2.)

8.2. Discussion

Close/distant general-purpose registers should be allocated to results rather than to variables. If a supercomputer architecture contains more general-purpose close/distant registers than active variables and temporary results, allocating some register to every active variable or temporary result is trivial. The number of close registers is limited, however, and they should be allocated to the variables and temporary results that are used in the most-critical paths. A result is either used in critical-path operations or it is not, but a variable can be used in critical-path operations during parts of its lifetime, and it can be used non-critical-path operations during other parts of its lifetime. Rather than tracking when a variable is used in critical-path operations and when it is used in non-critical-path operations, it is more straightforward to track the results that the variable contains. There are more results than variables, but result lifetimes are usually much shorter than variable lifetimes, so sets of

active results may be smaller than corresponding sets of active variables; this should simplify the register-allocation procedure.

With implicit addressing of registers based upon the history of use, as discussed in Chapter 7, the instruction-stream bandwidth required to address large sets of registers is small. With a close/distant organization of registers, execution speeds are as fast as if all registers are fast. Register allocation for close/distant registers, although different than traditional practice, is not unreasonably difficult. Scheduling for fast execution with a large set of registers may be easier than scheduling with a small set of registers. Therefore, it appears that the only cost of large sets of close/distant registers is the extra hardware required to implement them. As hardware cost is considered secondary to performance for supercomputers, the additional cost of large sets of close/distant registers is reasonable.

Generalized fast/bulk vector registers using PRAM should be implemented as monolithic integrated circuits. If the shift-registers, multiplexers, and counters are included in the chips, the number of pins drops dramatically. The performance limitation of a monolithic fast/bulk vector register is the speed with which elements of the shift register can be selected, and not the access-latency of the PRAM; probably a one-bit to four-bit slice of a vector register could be implemented in each chip. Vector registers that can support two vector reads at a time could also be implemented by including two sets of shift-registers, multiplexers, and counters. The availability of fast, monolithic vector registers will allow vector registers and vector instructions to be included in lower-cost computer architectures, and not just in supercomputers.

If PRAM chips are used to implement main memory, memory interference due to memory-bank busy times drops dramatically. Only simultaneous access of the same memory bank can cause an access conflict, and the conflict lasts only one clock tick (less, if the PRAM chips operate at a higher clock rate than the rest of the system). Considerably less memory

interleaving is required for good memory-system performance. Algorithms that access memory with irregular patterns can do so as fast as algorithms that access memory with regular access patterns. This reduction of access-pattern constraints should simplify algorithm development.

8.3. Conclusions

It is possible to increase the execution speed of supercomputers by using more registers, organized in a close/distant hierarchy, and by using generalized fast/bulk vector registers. With these small architectural changes, the average execution speed of the Cray-2 is increased by over 50%, for Livermore Kernels 1-14.

If enough registers are available to hold all the active temporary results, no time is wasted spilling results to memory, and operations can be scheduled to maximize execution speed rather than to keep register usage low. A close/distant organization of registers allows low-latency access of the results of critical-path operations. Implicit addressing reduces the instruction-stream bandwidth required to address many registers.

Generalized fast/bulk vector registers provide hardware support for long vectors, they allow full chaining of vector instructions, and they reduce the hardware requirements of vector registers. The lower-latency access reduces the execution time for short vectors, and the higher-bandwidth access supports full chaining of vector instructions, which reduces the execution time for all vectors. The smaller hardware requirements for each vector register allow either more vector registers or longer vector registers.

These techniques can also be applied to multiprocessors and general-purpose computers.

The communication buffers between multiprocessors are used like vector registers, except that one processor writes to the buffer, and a different processor reads from the buffer. Short messages should be transmitted with low latency, but long messages may need to be

stored in the buffer for long periods of time. Thus, low latency and high capacity are needed: these properties are precisely what fast/bulk vector registers provide. The buffers should be large, to allow for messages several megabytes long, and they should have the same width as the machine's word-size. For sending messages to several destinations at a time, the buffers could be constructed to support several vector reads at a time, as discussed in section 5.2.2.2.

General-purpose computers often support some overlap of execution, although not as much as supercomputers do. With more overlap of execution, more registers are necessary to hold the temporary results, but the generally-serial programs executed by general-purpose computers require low-latency access of the temporary results. These requirements are supported by close/distant register organizations. Just as it does for supercomputers, implicit addressing reduces the instruction-stream bandwidth required for general-purpose computers to address many registers.

8.4. Future Work

Livermore Kernels 1-14,⁶⁷ while representative of some scientific programs, do not represent all important programs. In particular, they contain no conditional branches, nor do they contain many operations besides loads, stores, adds, and multiplies. It would be useful to analyze programs that represent other workloads.

Design tradeoffs for pipelines with limited I/O rates are only discussed briefly in Chapter 4. The subject of non-linear pipelines is a rich one, which deserves a more thorough investigation. In particular, the relative merits of non-linear pipelines with fast clocks and short-stage pipelines with slow clocks should be studied, with an emphasis on single-chip implementations.

APPENDIX A

References

References

1. R.D. Acosta, J. Kjelstrup, and H.C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions on Computers* **C-35**(9) pp. 815-828 (September 1986).
2. T. Agerwala, "Microprogram Optimization: A Survey," *IEEE Transactions on Computers* **C-25**(9) pp. 11-17 (1976).
3. D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development*, pp. 8-24 (January 1967).
4. S.F. Anderson, J.G. Earle, R.E. Goldschmidt, and D.M. Powers, "The IBM System/360 Model 91: Floating Point Execution Unit," *IBM Journal of Research and Development*, pp. 34-53 (January 1967).
5. S. Arya, "Optimal Instruction Scheduling for a Class of Vector Processors: an Integer Programming Approach," PhD Dissertation: CRL-TR-19-83, University of Michigan Computing Research Laboratory, Ann Arbor (April 1983).
6. S. Arya, "An Optimal Instruction-Scheduling Model for a Class of Vector Processors," *IEEE Transactions on Computers* **C-34**(11) pp. 981-995 (November 1985).
7. D.H. Bailey, "Vector Computer Memory Bank Contention," *IEEE Transactions on Computers* **C-36**(3) pp. 293-298 (March 1987).

8. E.R. Berlekamp, "Some Observations on the Art of Writing Fast Loops for the CDC 6600," Unpublished Memo, Bell Telephone Laboratories (circa 1969).
9. E.R. Berlekamp, "Speedup for CDC 6600," Unpublished Memo, Bell Telephone Laboratories (circa 1970).
10. E.R. Berlekamp, "Galois Field Computer," U.S. Patent 4,162,480 (July 24 1979).
11. B.A. Blaauw, *Digital System Implementation*, Prentice-Hall, Englewood Cliffs, N.J. (1976).
12. E. Bloch and D. Galage, "Component Progress: Its Effect on High-Speed Computer Architecture and Machine Organization," *Computer*, pp. 64-76 (April 1978).
13. E.O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).
14. I.Y. Bucher, "The Computational Speed of Supercomputers," *Proc. 1983 Sigmetrics Conference*, pp. 151-165 (1983).
15. T. Cheung and J.E. Smith, "A simulation Study of the Cray X-MP Memory System," *IEEE Transactions on Computers* **C-35**(7) pp. 613-632 (July 1986).
16. F.C. Chow, "A Portable Machine-Independent Global Optimizer: Design and Measurements," PhD Dissertation: Computer Systems Laboratory Technical Note No. 83-254, Stanford University, Stanford, California (December 1983).
17. W.J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs (1980).
18. Control Data Corporation, *Control Data 7600 Computer System: Preliminary Reference Manual*, Control Data Corporation, Minneapolis (circa 1970).
19. L.W. Cotten, "Circuit Implementation of High-Speed Pipeline Systems," *AFIPS Proc. Fall Joint Computer Conference*, pp. 489-504 (1965).

20. L.W. Cotten, "Maximum-Rate Pipeline Systems," *AFIPS Proc. Spring Joint Computer Conference*, pp. 581-586 (1969).
21. Cray Research, Inc., "Cray-1 Computer Systems: Cray-1 S Series Hardware Reference Manual," HR-0808, Mendota Heights, Minn. (1981).
22. Cray Research, Inc., "Cray Computer Systems: Cray X-MP Model 48 Mainframe Reference Manual," HR-0097, Mendota Heights, Minn. (1984).
23. Cray Research, Inc., "Cray Computer Systems: Cray-2 Hardware Reference Manual," HR-2000, Mendota Heights, Minn. (1985).
24. Cray Research, Inc., "Civic Version 131e Cray-1 FORTRAN Compiler," Livermore, California (1986).
25. H.J. Curnow and B.A. Wichman, "A Synthetic Benchmark," *Computer Journal* **19**(1)(February 1976).
26. Denelcor, Inc., "HEP Hardware Reference Manual," 9000003, Denver (1982).
27. M. Dippe and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics* **18**(3) pp. 149-158 (July 1984).
28. D.R. Ditzel and H.R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 48-56 (March 1982).
29. J.J. Dongarra, *Linpack*, Argonne National Laboratory, Argonne (March 1978).
30. J.J. Dongarra and A. Hinds, "Comparison of the Cray X-MP-4, Fujitsu VP-200, and Hitachi S-810/20: An Argonne Perspective," ANL 85-19, Argonne National Laboratory (1985).
31. J.P. Eckert, "Univac-Larc, the Next Step in Computer Design," *Proc. EJCC*, pp. 16-20 (1956).

32. J.P. Eckert, J.C. Chu, A.B. Tonik, and W.F. Schmitt, "Design of UNIVAC-LARC System: I," *Proc. EJCC*, pp. 59-65 (1959).
33. ETA Systems, Inc., "Mainframe Subsystem Equipment Specification," 003106, St. Paul, Minn. (February 27, 1987).
34. ETA Systems, Inc., "Mainframe Subsystem Instruction Specification for the ETA¹⁰," 000211, St. Paul, Minn. (June 4, 1987).
35. Fairchild Camera and Instrument Corporation, *F100K ECL Data Book*, Fairchild Camera and Instrument Corporation, Mountain View, California (1982).
36. M.J. Flynn, J.J. Johnson, and S.P. Wakefield, "On Instruction Sets and Their Formats," *IEEE Transactions on Computers* **C-34**(3) pp. 242-254 (March 1985).
37. M.J. Flynn, C.L. Mitchell, and J.M. Mulder, "And Now a Case for More Complex Instruction Sets," *IEEE Computer* **20**(9) pp. 71-83 (September 1987).
38. C.C. Foster and E. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution," *IEEE Transactions on Computers* **C-21**(12) pp. 1411-1415 (December 1972).
39. S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood, Chichester, England (1982).
40. T. Furukatsu, T. Watanabe, and Ryoze Kondo, "NEC Supercomputer SX System (English Translation)," *Nikkei Electronics* **11**(19) pp. 237-272 (1984).
41. B.S. Garbow, *Eispack*, Argonne National Laboratory, Argonne (1983).
42. M.R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco (1979).
43. GigaBit Logic, *12G014 256x4 Static RAM Data Sheet*, GigaBit Logic, Newbury Park (September 1987).

44. M.J. Gonzalez, "Deterministic Processor Scheduling," *Computing Surveys* **9**(3) pp. 173-204 (1977).
45. A. Graham and S. Sando, "Pipelined Static RAM Endows Cache Memories with 1-ns Speed," *Electronic Design*, Hayden Publishing Co., Inc., (December 27, 1984).
46. T.G. Hallin and M.J. Flynn, "Pipelining of Arithmetic Functions," *IEEE Transactions on Computers* **C-21**(8) pp. 880-886 (August 1972).
47. D. Huff, *How to Lie with Statistics*, W.W. Norton and Company, New York (1954).
48. D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. IRE* **40** pp. 1098-1101 (September 1952).
49. A. Jippo, H. Izumisawa, H. Matsumoto, T. Kawamura, T. Furui, and Y. Izutani, "The Supercomputer SX System: Hardware," *Proc. Second International Conference on Supercomputing*, pp. 57-64 (1987).
50. J.M. Kats, R. Llurba, and A.J. van der Steen, "Experiences with the First Generation Japanese Supercomputers: Fujitsu VP-200, Hitachi S810/20 and NEC SX-2," *Proc. Second International Conference on Supercomputing*, pp. 159-167 (1987).
51. R.M. Keller, "Look-Ahead Processors," *Computing Surveys* **7**(4) pp. 177-195 (1975).
52. P.M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, San Francisco (1981).
53. W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on Computers* **C-24**(12) pp. 1235-1238 (1975).
54. J.S. Kolodzey, "Cray-1 Computer Technology," *IEEE Transactions on Components, Hybrids, and Manufacturing Technology* **CHMT-4**(2) pp. 181-186 (1981).

55. D.J. Kuck, *The Structure of Computers and Computations: Volume One*, Wiley, New York (1978).
56. H.T. Kung, "Memory Requirements for Balanced Computer Architectures," *Proc. 13th Ann. Symp. Computer Architecture*, pp. 49-54 (1986).
57. S.R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers," *Proc. 13th Ann. Symp. Computer Architecture*, pp. 404-413 (1986).
58. Los Alamos National Laboratory, *BMK*, Los Alamos National Laboratory, Los Alamos (circa 1985).
59. Los Alamos National Laboratory, *Simple*, Los Alamos National Laboratory, Los Alamos (circa 1985).
60. J.I. Lambiotte, Jr. and R.G. Voight, "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer," *ACM Transactions on Mathematical Software* **1**(4) pp. 308-329 (December 1975).
61. D. Landskov, S. Davidson, B. Shriver, and P. Mallet, "Local Microcode Compaction Techniques," *Computing Surveys* **12**(3) pp. 261-294 (1980).
62. J.L. Larson, "Multitasking on the Cray X-MP-2 Multiprocessor," *Computer Magazine*, pp. 62-69 (July 1984).
63. N.R. Lincoln, "It's Really Not As Much Fun Building A Supercomputer As It Is Simply Inventing One," *Proc. Symp. on High Speed Computer And Algorithm Organization*, pp. 3-11 Academic Press, (1977).
64. N.R. Lincoln, "Technology and Design Trade Offs in the Creation of a Modern Super-computer," *IEEE Transactions on Computers* **C-31**(5) pp. 363-376 (May 1982).
65. H. Lukoff, L.M. Spandorfer, and F.F. Lee, "Design of Univac-LARC System: II," *Proc. EJCC*, pp. 66-74 (1959).

66. W.D. Maurer, "A Theory of Computer Instructions," *JACM* **13**(2) pp. 226-235 (April 1966).
67. F.H. McMahon, "LLNL FORTRANS KERNELS: MFLOPS," Lawrence Livermore National Laboratory (March 1984).
68. R.M. Meade, "On Memory System Design," *Proc. AFIPS FJCC* **37** pp. 33-43 (November 1970).
69. Monolithic Memories, Inc., *Monolithic Memories LSI Databook Sixth Edition*, Monolithic Memories, Inc., Santa Clara, California (1985).
70. Motorola Incorporated, *MECL Device Data*, Motorola Incorporated, Phoenix, Arizona (1985).
71. S.S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).
72. H.L. Nelson, "Timing Codes on the Cray-1: Principles and Applications," UCID-30179, Lawrence Livermore National Laboratories (May 1981).
73. B. Nour-Omid and B.N. Parlett, "Element Preconditioning," PAM-103, Center for Pure and Applied Mathematics University of California, Berkeley (October 1982).
74. W. Oed and O. Lange, "On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems," *IEEE Transactions on Computers* **C-34**(10) pp. 949-957 (October 1985).
75. C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs (1982).
76. Y.N. Patt, W.W. Hwu, and M.C. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," *Proc. 18th International Microprogramming Workshop*, (December 1985).

77. A.V. Pohm and O.P. Agrawal, *High-Speed Memory Systems*, Reston (Prentice-Hall), Reston, Virginia (1983).
78. G. Polya, *How to Solve It: a New Aspect of Mathematical Method*, Princeton University Press, Princeton, New Jersey (1973).
79. W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge (1986).
80. C.V. Ramamoorthy and H.F. Li, "Pipeline Architecture," *Computer Surveys* **9**(1) pp. 61-102 (March 1977).
81. B.R. Rau and G.E. Rossman, "The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units," *Proc. 4th Ann. Symp. Computer Architecture*, pp. 80-89 (1977).
82. B.R. Rau, "Program Behavior and the Performance of Interleaved Memories," *IEEE Transactions on Computers* **C-28**(3) pp. 191-199 (March 1979).
83. B.R. Rau, C.D. Glaeser, and R. L. Picard, "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support," *Proc. 9th Ann. Symp. Computer Architecture*, pp. 131-139 (1982).
84. D.R. Resnick, "Hardware Technologies for the ETA 10," *Proc. Second International Conference on Supercomputing*, pp. 153-157 (1987).
85. E.M. Riseman and C.C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *Transactions on Computers* **C-21**(12) pp. 1405-1411 (December 1972).
86. R.M. Russell, "The Cray-1 Computer System," *CACM* **21**(1) pp. 63-72 (January 1978).
87. R.D. Shapiro, "Scheduling Coupled Tasks," *Naval Research Logistics Quarterly* **27** pp. 489-498 (1980).

88. L.E. Shar, "Design and Scheduling of Statically Configured Pipelines," Digital Systems Lab Report SU-SEL-72-042, Stanford University, Stanford, California (September 1972).
89. Signetics Corporation, *ECL 10K/100K Data Manual*, Signetics Corporation, Sunnyvale, California (1986).
90. R.L. Sites, "How to Use 1000 Registers," *Caltech Conference on VLSI*, pp. 527-532 (January 1979).
91. A.J. Smith, "Cache Memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (September 1982).
92. J.E. Smith, "Decoupled Access/Execute Computer Architectures," *Proc. Ninth Ann. Symp. on Computer Architecture* **9** pp. 112-119 (April 1982).
93. J.E. Smith, S. Weiss, and N.Y. Pang, "A Simulation study of Decoupled Architecture Computers," *IEEE Transactions on Computers* **C-35**(8) pp. 692-702 (August 1986).
94. E.E. Swartzlander, Jr., *Computer Arithmetic*, Dowden, Hutchinson & Ross, Inc., Stroudsburg, Pennsylvania (1980).
95. J.A. Swensen, "An Experimental Graphics Terminal," Masters Report, University of California at Berkeley, Berkeley, California (December 1982).
96. J.A. Swensen and Y.N. Patt, "Fast Temporary Storage for Serial and Parallel Execution," *Proc. 14th Annual Symp. on Computer Architecture*, pp. 35-43 (June 1987).
97. S.H. Takahashi, K. Nishino, and K. Fuchi, "System Design of the ETL MK-6 Computer," *Proc. 1962 IFIPS Congress*, pp. 690-693 North-Holland, (1963).
98. A.T. Thomas, "Scheduling of Multiconfigurible Pipelines," Coordinated Science Laboratory, Report T-78, University of Illinois, Urbana, Illinois (May 1979).

99. J.E. Thornton, "Parallel Operation in the Control Data 6600," *Proc. SJCC*, pp. 33-39 (1964).
100. J.E. Thornton, *Design of a Computer: The Control Data 6600*, Glenview, Illinois (1970).
101. G.S. Tjaden and M.J. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers* **C-19**(10) pp. 889-895 (December 1970).
102. G.S. Tjaden and M.J. Flynn, "Representation of Concurrency with Ordering Matrices," *IEEE Transactions on Computers* **C-22**(8) pp. 752-761 (August 1973).
103. R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Dev.*, pp. 25-33 (January 1967).
104. M. Tsukakoshi, H. Katayama, K. Abe, and K. Yamamoto, "The Supercomputer SX System: FORTRAN77/SX and Support Tools," *Proc. Second International Conference on Supercomputing*, pp. 72-79 (1987).
105. A.A. Vacca, "Considerations for High Performance LSI Applications," *Proc. 18th IEEE Computer Society International Conference*, pp. 278-284 (1979).
106. T. Watanabe, T. Furukatsu, R. Kondo, T. Kawamura, and Y. Izutani, "The Supercomputer SX System: An Overview," *Proc. Second International Conference on Supercomputing*, pp. 51-56 (1987).
107. W.J. Watson, "The TI ASC: A Highly Modular and Flexible Super Computer Architecture," *Proc. AFIPS FJCC*, pp. 221-228 (1972).
108. R.P. Weicker, "Dhrystone," *CACM* **27**(10) p. 1013 (October 1984).
109. S. Weiss and J.E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Transactions on Computers* **C-33**(11) pp. 1013-1022 (1984).
110. S. Weiss and J.E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," *Proceedings of the Second International Conference on Architectural*

Support for Programming Languages and Operating Systems, pp. 105–109 (October 1987).

111. M. Yamamoto, M. Aizawa, and K. Ikumi, "The Supercomputer SX System: Operating System," *Proc. Second International Conference on Supercomputing*, pp. 65–71 (1987).

APPENDIX B

Livermore Kernel Descriptions

Livermore Kernel 1: Hydro Excerpt

for k = 1 to n
 $x[k] = q + y[k] * (r * z[k+10] + t * z[k+11])$

With $n = 400$, there are 3204 operations in six partitions.

Livermore Kernel 2: MLR, Inner Product

for k = 1 to n * 5 by 5
 $tp[k] = z[k] * x[k] + z[k+1] * x[k+1] + z[k+2] * x[k+2]$
 $+ z[k+3] * x[k+3] + z[k+4] * x[k+4]$

With $n = 40$, there are 800 operations in six partitions.

Livermore Kernel 3: Inner Product

for k = 1 to n
 $q = q + z[k] * x[k]$

With $n = 1024$, there are 4098 operations in 14 partitions.

Livermore Kernel 4: Banded Linear Equations

for l = 7 to 107 by 50
for j = 1 to n
 $x[l-1] = x[l-1] - x[l+j-1] * y[j]$

With $n = 128$, there are 1542 operations in 11 partitions.

Livermore Kernel 5: Tri-Diagonal Elimination, Below Diagonal

for i = 2 to n
 $x[i] = z[i] * (y[i] - x[i-1])$

With $n = 997$, there are 4981 operations in 1994 partitions.

Livermore Kernel 6: Tri-Diagonal Elimination, Above Diagonal

for i = n-1 downto 1
 $x[i] = x[i] - z[i] * x[i+1]$

With $n = 997$, there are 4986 operations in 1996 partitions.

Livermore Kernel 7: Equation of State Excerptfor $m = 1$ to n

$$\begin{aligned}
 x[m] &= u[m] + r * (z[m] + r * y[m]) \\
 &+ t * (u[m+3] + r * (u[m+2] + r * u[m+1])) \\
 &+ t * (u[m+6] + r * (u[m+5] + r * u[m+4]))
 \end{aligned}$$

With $n = 120$, there are 2408 operations in 12 partitions.**Livermore Kernel 8: PDE Integration**for $kx = 2$ to 3for $ky = 2$ to n

$$\begin{aligned}
 du1[ky] &= u1[kx,ky+1,nl1] - u1[kx,ky-1,nl1] \\
 du2[ky] &= u2[kx,ky+1,nl1] - u2[kx,ky-1,nl1] \\
 du3[ky] &= u3[kx,ky+1,nl1] - u3[kx,ky-1,nl1] \\
 u1[kx,ky,nl2] &= u1[kx,ky,nl1] \\
 &+ a11 * du1[ky] + a12 * du2[ky] + a13 * du3[ky] \\
 &+ sig * (u1[kx+1,ky,nl1] - 2 * u1[kx,ky,nl1] + u1[kx-1,ky,nl1]) \\
 u2[kx,ky,nl2] &= u2[kx,ky,nl1] \\
 &+ a21 * du1[ky] + a22 * du2[ky] + a23 * du3[ky] \\
 &+ sig * (u2[kx+1,ky,nl1] - 2 * u2[kx,ky,nl1] + u2[kx-1,ky,nl1]) \\
 u3[kx,ky,nl2] &= u3[kx,ky,nl1] \\
 &+ a31 * du1[ky] + a32 * du2[ky] + a33 * du3[ky] \\
 &+ sig * (u3[kx+1,ky,nl1] - 2 * u3[kx,ky,nl1] + u3[kx-1,ky,nl1])
 \end{aligned}$$

With $n = 20$, there are 2062 operations in seven partitions.**Livermore Kernel 9: Integrate Predictors**for $i = 1$ to n

$$\begin{aligned}
 px[1,i] &= bm28 * px[13,i] + bm27 * px[12,i] + bm26 * px[11,i] \\
 &+ bm25 * px[10,i] + bm24 * px[9,i] + bm23 * px[8,i] \\
 &+ bm22 * px[7,i] + c0 * (px[5,i] + px[6,i]) + px[3,i]
 \end{aligned}$$

With $n = 100$, there are 2808 operations in seven partitions.

Livermore Kernel 10: Difference Predictors

```

for i = 1 to n
  ar = cx[5,i]
  br = ar - px[5,i]
  px[5,i] = ar
  cr = br - px[6,i]
  px[6,i] = br
  ar = cr - px[7,i]
  px[7,i] = cr
  br = ar - px[8,i]
  px[8,i] = ar
  cr = br - px[9,i]
  px[9,i] = br
  ar = cr - px[10,i]
  px[10,i] = cr
  br = ar - px[11,i]
  px[11,i] = ar
  cr = br - px[12,i]
  px[12,i] = br
  px[14,i] = cr - px[13,i]
  px[13,i] = cr

```

With $n = 100$, there are 2900 operations in 11 partitions.

Livermore Kernel 11: First Sum

```

for k = 2 to n
  x[k] = x[k-1] + y[k]

```

With $n = 999$, there are 2998 operations in 1001 partitions.

Livermore Kernel 12: First Diff.

```

for k = 1 to n
  x[k] = y[k+1] - y[k]

```

With $n = 1000$, there are 3001 operations in three partitions.

Livermore Kernel 13: 2-D Particle Pusher

```

for ip = 1 to n
  i1 = p[1,ip]
  j1 = p[2,ip]
  p[3,ip] = p[3,ip] + b[i1,j1]
  p[4,ip] = p[4,ip] + c[i1,j1]
  p[1,ip] = p[1,ip] + p[3,ip]
  p[2,ip] = p[2,ip] + p[4,ip]
  i2 = p[1,ip]
  j2 = p[2,ip]
  p[1,ip] = p[1,ip] + y[i2+32]
  p[2,ip] = p[2,ip] + z[j2+32]
  i2 = i2 + e[i2+32]
  j2 = j2 + f[j2+32]
  h[i2,j2] = h[i2,j2] + 1.0

```

With $n = 128$, there are 3456 operations in 390 partitions.

Livermore Kernel 14: 1-D Particle Pusher

```

for k = 1 to n
  ix = grd[k]
  xi = ix
  vx[k] = vx[k] + ex[ix] + (xx[k]-xi) + dex[ix]
  xx[k] = xx[k] + vx[k] + flx
  ir = xx[k]
  ri = ir
  rx1 = xx[k] - ri
  ir = ir AND 63
  xx[k] = ri + rx1
  rh[ir] = rh[ir] + 1.0 - rx1
  rh[ir+1] = rh[ir+1] + rx1

```

With $n = 150$, there are 3601 operations in 606 partitions.

APPENDIX C

Execution Time Predictions and Simulations

The upper and lower bounds on execution time are computed from Theorems 3.6 and 3.7, and are displayed with the simulation times. Root mean square differences between bounds and simulation times for all values of the parameters are shown below each table.

Table C.1: Execution Time Simulations and Bounds for Livermore Kernel 1, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds				
				Upper	(diff)	Lower	(diff)	
Cray-1	1	0	3216	3252	1.1%	3206	-0.3%	
	2		1614	1651	2.3%	1604	-0.6%	
	4		813	850	4.6%	803	-1.2%	
	8		413	450	9.0%	403	-2.4%	
	4	1	815	856	5.0%	804	-1.3%	
	8		415	456	9.9%	404	-2.7%	
	2	1	1616	1657	2.5%	1605	-0.7%	
	4		817	862	5.5%	805	-1.5%	
	8		419	468	11.7%	406	-3.1%	
	1	1	0	3205	3210	0.2%	3205	0.0%
		2		1603	1609	0.4%	1603	0.0%
		4		802	808	0.7%	802	0.0%
8		402		408	1.5%	402	0.0%	
2	1	3207		3216	0.3%	3206	-0.0%	
	2	1605		1615	0.6%	1604	-0.1%	
	4	804		814	1.2%	803	-0.1%	
	8	404		414	2.5%	403	-0.2%	
4	1	3211		3228	0.5%	3208	-0.1%	
	2	1609		1627	1.1%	1606	-0.2%	
	4	808		826	2.2%	805	-0.4%	
	8	408		426	4.4%	405	-0.7%	
8	1	3219		3252	1.0%	3212	-0.2%	
	2	1617		1651	2.1%	1610	-0.4%	
	4	816		850	4.2%	809	-0.9%	
	8	416		450	8.2%	409	-1.7%	
16	1	3235	3300	2.0%	3220	-0.5%		
	2	1633	1699	4.0%	1618	-0.9%		
	4	832	898	7.9%	817	-1.8%		
	8	432	498	15.3%	417	-3.5%		
32	1	3267	3396	3.9%	3236	-0.9%		
	2	1665	1795	7.8%	1634	-1.9%		
	4	864	994	15.0%	833	-3.6%		
	8	464	594	28.0%	433	-6.7%		
64	1	3331	3588	7.7%	3268	-1.9%		
	2	1729	1987	14.9%	1666	-3.6%		
	4	928	1186	27.8%	865	-6.8%		
	8	507	786	55.0%	493	-2.8%		

RMS bound differences = 12.8% (upper) 2.2% (lower)

Table C.2: Execution Time Simulations and Bounds for Livermore Kernel 2, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds				
				Upper	(diff)	Lower	(diff)	
Cray-1	1	0	802	844	5.2%	802	0.0%	
	2		402	444	10.4%	402	0.0%	
	4		202	244	20.8%	202	0.0%	
	8		106	144	35.8%	103	-2.8%	
	4	1	203	250	23.2%	203	0.0%	
	8		110	150	36.4%	105	-4.5%	
	2	1	403	450	11.7%	403	0.0%	
	4		204	256	25.5%	204	0.0%	
	8		118	162	37.3%	109	-7.6%	
	1	1	0	801	806	0.6%	801	0.0%
		2		401	406	1.2%	401	0.0%
		4		201	206	2.5%	201	0.0%
8		101		106	5.0%	101	0.0%	
2	1	802		812	1.2%	802	0.0%	
	2	402		412	2.5%	402	0.0%	
	4	202		212	5.0%	202	0.0%	
	8	102		112	9.8%	102	0.0%	
4	1	804		824	2.5%	804	0.0%	
	2	404		424	5.0%	404	0.0%	
	4	204		224	9.8%	204	0.0%	
	8	104		124	19.2%	104	0.0%	
8	1	808	848	5.0%	808	0.0%		
	2	408	448	9.8%	408	0.0%		
	4	208	248	19.2%	208	0.0%		
	8	114	148	29.8%	111	-2.6%		
16	1	816	896	9.8%	816	0.0%		
	2	416	496	19.2%	416	0.0%		
	4	228	296	29.8%	222	-2.6%		
	8	144	196	36.1%	133	-7.6%		
32	1	832	992	19.2%	832	0.0%		
	2	456	592	29.8%	444	-2.6%		
	4	288	392	36.1%	266	-7.6%		
	8	232	292	25.9%	205	-11.6%		
64	1	912	1184	29.8%	888	-2.6%		
	2	576	784	36.1%	532	-7.6%		
	4	464	584	25.9%	410	-11.6%		
	8	424	484	14.2%	389	-8.3%		

RMS bound differences = 21.4% (upper) 4.1% (lower)

Table C.3: Execution Time Simulations and Bounds for Livermore Kernel 3, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	4120	4189	1.7%	4115	-0.1%
	2		2078	2141	3.0%	2073	-0.2%
	4		1060	1118	5.5%	1055	-0.5%
	8		554	607	9.6%	549	-0.9%
	4	1	1068	1132	6.0%	1062	-0.6%
	8		563	621	10.3%	557	-1.1%
	2	1	2085	2155	3.4%	2079	-0.3%
	4		1076	1146	6.5%	1069	-0.7%
	8		582	649	11.5%	574	-1.4%
1	1	0	4099	4112	0.3%	4099	0.0%
	2		2051	2064	0.6%	2051	0.0%
	4		1028	1041	1.3%	1028	0.0%
	8		517	530	2.5%	517	0.0%
2	1		4102	4126	0.6%	4101	-0.0%
	2		2056	2078	1.1%	2055	-0.0%
	4		1034	1055	2.0%	1033	-0.1%
	8		524	544	3.8%	523	-0.2%
4	1		4112	4154	1.0%	4109	-0.1%
	2		2068	2106	1.8%	2065	-0.1%
	4		1048	1083	3.3%	1045	-0.3%
	8		540	572	5.9%	537	-0.6%
8	1		4136	4210	1.8%	4129	-0.2%
	2		2096	2162	3.1%	2089	-0.3%
	4		1080	1139	5.5%	1073	-0.6%
	8		576	628	9.0%	569	-1.2%
16	1	4192	4322	3.1%	4177	-0.4%	
	2	2160	2274	5.3%	2145	-0.7%	
	4	1152	1251	8.6%	1137	-1.3%	
	8	656	740	12.8%	641	-2.3%	
32	1	4320	4546	5.2%	4289	-0.7%	
	2	2304	2498	8.4%	2273	-1.3%	
	4	1312	1475	12.4%	1281	-2.4%	
	8	832	964	15.9%	801	-3.7%	
64	1	4608	4994	8.4%	4545	-1.4%	
	2	2624	2946	12.3%	2561	-2.4%	
	4	1664	1923	15.6%	1601	-3.8%	
	8	1216	1412	16.1%	1153	-5.2%	

RMS bound differences = 7.7% (upper) 1.5% (lower)

Table C.4: Execution Time Simulations and Bounds for Livermore Kernel 4, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	1554	1615	3.9%	1547	-0.5%
	2		788	845	7.2%	784	-0.5%
	4		410	460	12.2%	405	-1.2%
	8		224	268	19.6%	219	-2.2%
	4	1	417	471	12.9%	411	-1.4%
	8		232	279	20.3%	226	-2.6%
	2	1	794	856	7.8%	789	-0.6%
	4		424	482	13.7%	417	-1.7%
	8		248	301	21.4%	240	-3.2%
1	1	0	1543	1553	0.6%	1543	0.0%
	2		772	783	1.4%	772	0.0%
	4		387	398	2.8%	387	0.0%
	8		196	206	5.1%	195	-0.5%
2	1		1544	1564	1.3%	1544	0.0%
	2		774	794	2.6%	774	0.0%
	4		391	409	4.6%	390	-0.3%
	8		201	217	8.0%	200	-0.5%
4	1		1548	1586	2.5%	1547	-0.1%
	2		782	816	4.3%	780	-0.3%
	4		402	431	7.2%	399	-0.7%
	8		214	239	11.7%	211	-1.4%
8	1	1564	1630	4.2%	1559	-0.3%	
	2	804	860	7.0%	798	-0.7%	
	4	428	475	11.0%	421	-1.6%	
	8	244	283	16.0%	237	-2.9%	
16	1	1608	1718	6.8%	1595	-0.8%	
	2	856	948	10.7%	842	-1.6%	
	4	488	563	15.4%	473	-3.1%	
	8	312	371	18.9%	297	-4.8%	
32	1	1712	1894	10.6%	1683	-1.7%	
	2	976	1124	15.2%	946	-3.1%	
	4	624	739	18.4%	593	-5.0%	
	8	464	547	17.9%	433	-6.7%	
64	1	1952	2246	15.1%	1891	-3.1%	
	2	1248	1476	18.3%	1186	-5.0%	
	4	928	1091	17.6%	865	-6.8%	
	8	800	899	12.4%	737	-7.9%	

RMS bound differences = 12.1% (upper) 2.9% (lower)

Table C.5: Execution Time Simulations and Bounds for Livermore Kernel 5, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds				
				Upper	(diff)	Lower	(diff)	
Cray-1	1	0	12963	26902	107.5%	12962	-0.0%	
	2		12962	24412	88.3%	12962	0.0%	
	4		12962	24163	86.4%	12962	0.0%	
	8		12962	24039	85.5%	12962	0.0%	
	4	1	14956	26157	74.9%	14956	0.0%	
	8		14956	26033	74.1%	14956	0.0%	
	2	1	14956	26406	76.6%	14956	0.0%	
	4		2	16950	28151	66.1%	16950	0.0%
	8		3	18944	30021	58.5%	18944	0.0%
1	1	0	4982	6975	40.0%	4982	0.0%	
	2		2492	4485	80.0%	2492	0.0%	
	4		1995	4236	112.3%	1995	0.0%	
	8		1995	4112	106.1%	1995	0.0%	
2	1		4984	8969	80.0%	4983	-0.0%	
	2		3989	6479	62.4%	3989	0.0%	
	4		3989	6230	56.2%	3989	0.0%	
	8		3989	6106	53.1%	3989	0.0%	
4	1		7978	12957	62.4%	7977	-0.0%	
	2		7977	10467	31.2%	7977	0.0%	
	4		7977	10218	28.1%	7977	0.0%	
	8		7977	10094	26.5%	7977	0.0%	
8	1		15954	20933	31.2%	15953	-0.0%	
	2		15953	18443	15.6%	15953	0.0%	
	4		15953	18194	14.0%	15953	0.0%	
	8		15953	18070	13.3%	15953	0.0%	
16	1	31906	36885	15.6%	31905	-0.0%		
	2	31905	34395	7.8%	31905	0.0%		
	4	31905	34146	7.0%	31905	0.0%		
	8	31905	34022	6.6%	31905	0.0%		
32	1	63810	68789	7.8%	63809	-0.0%		
	2	63809	66299	3.9%	63809	0.0%		
	4	63809	66050	3.5%	63809	0.0%		
	8	63809	65926	3.3%	63809	0.0%		
64	1	127618	132597	3.9%	127617	-0.0%		
	2	127617	130107	2.0%	127617	0.0%		
	4	127617	129858	1.8%	127617	0.0%		
	8	127617	129734	1.7%	127617	0.0%		

RMS bound differences = 55.9% (upper) 0.0% (lower)

Table C.6: Execution Time Simulations and Bounds for Livermore Kernel 6, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	12976	26928	107.5%	12975	-0.0%
	2		12975	24436	88.3%	12975	0.0%
	4		12975	24187	86.4%	12975	0.0%
	8		12975	24062	85.4%	12975	0.0%
	4	1	14971	26183	74.9%	14971	0.0%
	8		14971	26058	74.1%	14971	0.0%
	2	1	14971	26432	76.6%	14971	0.0%
	4		16967	28179	66.1%	16967	0.0%
	8		18963	30050	58.5%	18963	0.0%
	1		0	4987	6982	40.0%	4987
2	2495	4490		80.0%	2494	-0.0%	
4	1997	4241		112.4%	1997	0.0%	
8	1997	4116		106.1%	1997	0.0%	
2	1	0	4989	8978	80.0%	4988	-0.0%
	2		3993	6486	62.4%	3993	0.0%
	4		3993	6237	56.2%	3993	0.0%
	8		3993	6112	53.1%	3993	0.0%
4	1	0	7986	12970	62.4%	7985	-0.0%
	2		7985	10478	31.2%	7985	0.0%
	4		7985	10229	28.1%	7985	0.0%
	8		7985	10104	26.5%	7985	0.0%
8	1	0	15970	20954	31.2%	15969	-0.0%
	2		15969	18462	15.6%	15969	0.0%
	4		15969	18213	14.1%	15969	0.0%
	8		15969	18088	13.3%	15969	0.0%
16	1	0	31938	36922	15.6%	31937	-0.0%
	2		31937	34430	7.8%	31937	0.0%
	4		31937	34181	7.0%	31937	0.0%
	8		31937	34056	6.6%	31937	0.0%
32	1	0	63874	68858	7.8%	63873	-0.0%
	2		63873	66366	3.9%	63873	0.0%
	4		63873	66117	3.5%	63873	0.0%
	8		63873	65992	3.3%	63873	0.0%
64	1	0	127746	132730	3.9%	127745	-0.0%
	2		127745	130238	2.0%	127745	0.0%
	4		127745	129989	1.8%	127745	0.0%
	8		127745	129864	1.7%	127745	0.0%

RMS bound differences = 55.9% (upper) 0.0% (lower)

Table C.7: Execution Time Simulations and Bounds for Livermore Kernel 7, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	2410	2517	4.4%	2410	0.0%
			1206	1316	9.1%	1206	0.0%
			604	715	18.4%	604	0.0%
			303	415	37.0%	303	0.0%
	4	1	605	727	20.2%	605	0.0%
			304	427	40.5%	304	0.0%
	8	1	1207	1328	10.0%	1207	0.0%
			606	739	21.9%	606	0.0%
			306	451	47.4%	306	0.0%
	1	0	2409	2420	0.5%	2409	0.0%
			1205	1219	1.2%	1205	0.0%
			603	618	2.5%	603	0.0%
302			318	5.3%	302	0.0%	
2	0	2410	2432	0.9%	2410	0.0%	
		1206	1231	2.1%	1206	0.0%	
		604	630	4.3%	604	0.0%	
		303	330	8.9%	303	0.0%	
4	0	2412	2456	1.8%	2412	0.0%	
		1208	1255	3.9%	1208	0.0%	
		606	654	7.9%	606	0.0%	
		305	354	16.1%	305	0.0%	
8	0	2416	2504	3.6%	2416	0.0%	
		1212	1303	7.5%	1212	0.0%	
		610	702	15.1%	610	0.0%	
		309	402	30.1%	309	0.0%	
16	0	2424	2600	7.3%	2424	0.0%	
		1220	1399	14.7%	1220	0.0%	
		618	798	29.1%	618	0.0%	
		320	498	55.6%	319	-0.3%	
32	0	2440	2792	14.4%	2440	0.0%	
		1236	1591	28.7%	1236	0.0%	
		640	990	54.7%	638	-0.3%	
		400	690	72.5%	399	-0.3%	
64	0	2472	3176	28.5%	2472	0.0%	
		1280	1975	54.3%	1276	-0.3%	
		799	1374	72.0%	798	-0.1%	
		784	1074	37.0%	783	-0.1%	

RMS bound differences = 29.5% (upper) 0.1% (lower)

Table C.8: Execution Time Simulations and Bounds for Livermore Kernel 8, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds					
				Upper	(diff)	Lower	(diff)		
Cray-1	1	0	2064	2117	2.6%	2064	0.0%		
	2		1033	1087	5.2%	1033	0.0%		
	4		518	572	10.4%	518	0.0%		
	8		260	315	21.2%	260	0.0%		
	4	1	519	579	11.6%	519	0.0%		
	8		261	322	23.4%	261	0.0%		
	2	1	1034	1094	5.8%	1034	0.0%		
	4		520	586	12.7%	520	0.0%		
	8		263	336	27.8%	263	0.0%		
	1	1	0	2063	2069	0.3%	2063	0.0%	
				2	1032	1039	0.7%	1032	0.0%
				4	517	524	1.4%	517	0.0%
8				259	267	3.1%	259	0.0%	
2	1	2064		2076	0.6%	2064	0.0%		
		2		1033	1046	1.3%	1033	0.0%	
		4		518	531	2.5%	518	0.0%	
		8		260	274	5.4%	260	0.0%	
4	1	2066		2090	1.2%	2066	0.0%		
		2		1035	1060	2.4%	1035	0.0%	
		4		520	545	4.8%	520	0.0%	
		8		262	288	9.9%	262	0.0%	
8	1	2070	2118	2.3%	2070	0.0%			
		2	1039	1088	4.7%	1039	0.0%		
		4	524	573	9.4%	524	0.0%		
		8	266	316	18.8%	266	0.0%		
16	1	2078	2174	4.6%	2078	0.0%			
		2	1047	1144	9.3%	1047	0.0%		
		4	532	629	18.2%	532	0.0%		
		8	276	372	34.8%	274	-0.7%		
32	1	2094	2286	9.2%	2094	0.0%			
		2	1063	1256	18.2%	1063	0.0%		
		4	552	741	34.2%	548	-0.7%		
		8	326	484	48.5%	294	-9.8%		
64	1	2126	2510	18.1%	2126	0.0%			
		2	1103	1480	34.2%	1095	-0.7%		
		4	652	965	48.0%	588	-9.8%		
		8	494	708	43.3%	463	-6.3%		

RMS bound differences = 19.6% (upper) 2.5% (lower)

Table C.9: Execution Time Simulations and Bounds for Livermore Kernel 9, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds				
				Upper	(diff)	Lower	(diff)	
Cray-1	1	0	2810	2867	2.0%	2810	0.0%	
	2		1406	1464	4.1%	1406	0.0%	
	4		704	762	8.2%	704	0.0%	
	8		353	412	16.7%	353	0.0%	
	4	1	705	769	9.1%	705	0.0%	
	8		354	419	18.4%	354	0.0%	
	2	1	1407	1471	4.5%	1407	0.0%	
	4		2	706	776	9.9%	706	0.0%
	8		3	356	433	21.6%	356	0.0%
	1	1	0	2809	2815	0.2%	2809	0.0%
		2		1405	1412	0.5%	1405	0.0%
		4		703	710	1.0%	703	0.0%
8		352		360	2.3%	352	0.0%	
2	1	2810		2822	0.4%	2810	0.0%	
	2	1406		1419	0.9%	1406	0.0%	
	4	704		717	1.8%	704	0.0%	
	8	353		367	4.0%	353	0.0%	
4	1	2812		2836	0.9%	2812	0.0%	
	2	1408		1433	1.8%	1408	0.0%	
	4	706		731	3.5%	706	0.0%	
	8	355		381	7.3%	355	0.0%	
8	1	2816		2864	1.7%	2816	0.0%	
	2	1412		1461	3.5%	1412	0.0%	
	4	710		759	6.9%	710	0.0%	
	8	359		409	13.9%	359	0.0%	
16	1	2824	2920	3.4%	2824	0.0%		
	2	1420	1517	6.8%	1420	0.0%		
	4	718	815	13.5%	718	0.0%		
	8	374	465	24.3%	371	-0.8%		
32	1	2840	3032	6.8%	2840	0.0%		
	2	1436	1629	13.4%	1436	0.0%		
	4	748	927	23.9%	741	-0.9%		
	8	429	577	34.5%	410	-4.4%		
64	1	2872	3256	13.4%	2872	0.0%		
	2	1496	1853	23.9%	1482	-0.9%		
	4	858	1151	34.1%	819	-4.5%		
	8	584	801	37.2%	532	-8.9%		

RMS bound differences = 14.6% (upper) 1.8% (lower)

Table C.10: Execution Time Simulations and Bounds for Livermore Kernel 10, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	2902	3007	3.6%	2902	0.0%
	2		1452	1557	7.2%	1452	0.0%
	4		727	832	14.4%	727	0.0%
	8		365	470	28.8%	365	0.0%
	4	1	728	843	15.8%	728	0.0%
	8	366	481	31.4%	366	0.0%	
	2	1	1453	1568	7.9%	1453	0.0%
	4	2	729	854	17.1%	729	0.0%
	8	3	368	503	36.7%	368	0.0%
1	1	0	2901	2911	0.3%	2901	0.0%
	2		1451	1461	0.7%	1451	0.0%
	4		726	736	1.4%	726	0.0%
	8		364	374	2.7%	364	0.0%
2	1		2902	2922	0.7%	2902	0.0%
	2		1452	1472	1.4%	1452	0.0%
	4		727	747	2.8%	727	0.0%
	8		365	385	5.5%	365	0.0%
4	1		2904	2944	1.4%	2904	0.0%
	2		1454	1494	2.8%	1454	0.0%
	4		729	769	5.5%	729	0.0%
	8		367	407	10.9%	367	0.0%
8	1	2908	2988	2.8%	2908	0.0%	
	2	1458	1538	5.5%	1458	0.0%	
	4	733	813	10.9%	733	0.0%	
	8	371	451	21.6%	371	0.0%	
16	1	2916	3076	5.5%	2916	0.0%	
	2	1466	1626	10.9%	1466	0.0%	
	4	741	901	21.6%	741	0.0%	
	8	379	539	42.2%	379	0.0%	
32	1	2932	3252	10.9%	2932	0.0%	
	2	1482	1802	21.6%	1482	0.0%	
	4	757	1077	42.3%	757	0.0%	
	8	409	715	74.8%	395	-3.4%	
64	1	2964	3604	21.6%	2964	0.0%	
	2	1514	2154	42.3%	1514	0.0%	
	4	817	1429	74.9%	789	-3.4%	
	8	729	1067	46.4%	717	-1.6%	

RMS bound differences = 26.1% (upper) 0.8% (lower)

Table C.11 Execution Time Simulations and Bounds for Livermore Kernel 11, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	6009	13995	132.9%	6008	-0.0%
	2		6008	12497	108.0%	6008	0.0%
	4		6008	12247	103.8%	6008	0.0%
	8		6008	12122	101.8%	6008	0.0%
	4	1	7009	13248	89.0%	7009	0.0%
	8		7009	13123	87.2%	7009	0.0%
	2	1	7009	13498	92.6%	7009	0.0%
	4		2	8010	14249	77.9%	8010
8	3		9011	15125	67.9%	9011	0.0%
1	1	0	2999	3999	33.3%	2999	0.0%
	2		1500	2501	66.7%	1500	0.0%
	4		1002	2251	124.7%	1002	0.0%
	8		1002	2126	112.2%	1002	0.0%
2	1		3000	5000	66.7%	3000	0.0%
	2		2003	3502	74.8%	2003	0.0%
	4		2003	3252	62.4%	2003	0.0%
	8		2003	3127	56.1%	2003	0.0%
4	1		4006	7002	74.8%	4005	-0.0%
	2		4005	5504	37.4%	4005	0.0%
	4		4005	5254	31.2%	4005	0.0%
	8		4005	5129	28.1%	4005	0.0%
8	1	8010	11006	37.4%	8009	-0.0%	
	2	8009	9508	18.7%	8009	0.0%	
	4	8009	9258	15.6%	8009	0.0%	
	8	8009	9133	14.0%	8009	0.0%	
16	1	16018	19014	18.7%	16017	-0.0%	
	2	16017	17516	9.4%	16017	0.0%	
	4	16017	17266	7.8%	16017	0.0%	
	8	16017	17141	7.0%	16017	0.0%	
32	1	32034	35030	9.4%	32033	-0.0%	
	2	32033	33532	4.7%	32033	0.0%	
	4	32033	33282	3.9%	32033	0.0%	
	8	32033	33157	3.5%	32033	0.0%	
64	1	64066	67062	4.7%	64065	-0.0%	
	2	64065	65564	2.3%	64065	0.0%	
	4	64065	65314	1.9%	64065	0.0%	
	8	64065	65189	1.8%	64065	0.0%	

RMS bound differences = 63.2% (upper) 0.0% (lower)

Table C.12: Execution Time Simulations and Bounds for Livermore Kernel 12, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds				
				Upper	(diff)	Lower	(diff)	
Cray-1	1	0	3003	3020	0.6%	3003	0.0%	
	2		1503	1520	1.1%	1503	0.0%	
	4		753	770	2.3%	753	0.0%	
	8		378	395	4.5%	378	0.0%	
	4	1	754	773	2.5%	754	0.0%	
	8		379	398	5.0%	379	0.0%	
	2	1	1504	1523	1.3%	1504	0.0%	
	4		755	776	2.8%	755	0.0%	
	8		381	404	6.0%	381	0.0%	
1	1	0	3002	3004	0.1%	3002	0.0%	
			2	1502	1504	0.1%	1502	0.0%
			4	752	754	0.3%	752	0.0%
			8	377	379	0.5%	377	0.0%
2	1		3003	3007	0.1%	3003	0.0%	
			2	1503	1507	0.3%	1503	0.0%
			4	753	757	0.5%	753	0.0%
			8	378	382	1.1%	378	0.0%
4	1		3005	3013	0.3%	3005	0.0%	
			2	1505	1513	0.5%	1505	0.0%
			4	755	763	1.1%	755	0.0%
			8	380	388	2.1%	380	0.0%
8	1	3009	3025	0.5%	3009	0.0%		
		2	1509	1525	1.1%	1509	0.0%	
		4	759	775	2.1%	759	0.0%	
		8	384	400	4.2%	384	0.0%	
16	1	3017	3049	1.1%	3017	0.0%		
		2	1517	1549	2.1%	1517	0.0%	
		4	767	799	4.2%	767	0.0%	
		8	392	424	8.2%	392	0.0%	
32	1	3033	3097	2.1%	3033	0.0%		
		2	1533	1597	4.2%	1533	0.0%	
		4	783	847	8.2%	783	0.0%	
		8	408	472	15.7%	408	0.0%	
64	1	3065	3193	4.2%	3065	0.0%		
		2	1565	1693	8.2%	1565	0.0%	
		4	815	943	15.7%	815	0.0%	
		8	440	568	29.1%	440	0.0%	

RMS bound differences = 6.9% (upper) 0.0% (lower)

Table C.13: Execution Time Simulations and Bounds for Livermore Kernel 13, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds			
				Upper	(diff)	Lower	(diff)
Cray-1	1	0	3458	7087	104.9%	3458	0.0%
	2		2484	5551	123.5%	1849	-25.6%
	4		2484	4720	90.0%	1849	-25.6%
	8		2484	4181	68.3%	1849	-25.6%
	4	1	2874	5110	77.8%	2239	-22.1%
	8		2874	4571	59.0%	2239	-22.1%
	2	1	2874	5941	106.7%	2239	-22.1%
	4		3264	5500	68.5%	2629	-19.5%
	8		3654	5351	46.4%	3019	-17.4%
	1		0	3457	3846	11.3%	3457
	2	1729		2310	33.6%	1729	0.0%
	4	865		1479	71.0%	865	0.0%
8	433	940		117.1%	433	0.0%	
2	1	0	3458	4236	22.5%	3458	0.0%
	2		1730	2700	56.1%	1730	0.0%
	4		866	1869	115.8%	866	0.0%
	8		781	1330	70.3%	781	0.0%
4	1	0	3460	5016	45.0%	3460	0.0%
	2		1732	3480	100.9%	1732	0.0%
	4		1561	2649	69.7%	1561	0.0%
	8		1561	2110	35.2%	1561	0.0%
8	1	0	3464	6576	89.8%	3464	0.0%
	2		3121	5040	61.5%	3121	-0.0%
	4		3121	4209	34.9%	3121	0.0%
	8		3121	3670	17.6%	3121	0.0%
16	1	0	6243	9696	55.3%	6241	-0.0%
	2		6241	8160	30.7%	6241	0.0%
	4		6241	7329	17.4%	6241	0.0%
	8		6241	6790	8.8%	6241	0.0%
32	1	0	12483	15936	27.7%	12481	-0.0%
	2		12481	14400	15.4%	12481	0.0%
	4		12481	13569	8.7%	12481	0.0%
	8		12481	13030	4.4%	12481	0.0%
64	1	0	24963	28416	13.8%	24961	-0.0%
	2		24961	26880	7.7%	24961	0.0%
	4		24961	26049	4.4%	24961	0.0%
	8		24961	25510	2.2%	24961	0.0%

RMS bound differences = 62.9% (upper) 10.5% (lower)

Table C.14: Execution Time Simulations and Bounds for Livermore Kernel 14, with Differences Between Simulated and Theoretical Times.

Op times	Starts	Cost	Ticks	Bounds				
				Upper	(diff)	Lower	(diff)	
Cray-1	1	0	3794	10234	169.7%	3603	-5.0%	
	2		3793	8659	128.3%	2888	-23.9%	
	4		3793	7797	105.6%	2888	-23.9%	
	8		3793	7294	92.3%	2888	-23.9%	
	4	1	4399	8403	91.0%	3494	-20.6%	
	8		4399	7900	79.6%	3494	-20.6%	
	2	1	4399	9265	110.6%	3494	-20.6%	
	4		2	5005	9009	80.0%	4100	-18.1%
	8		3	5611	9112	62.4%	4706	-16.1%
1	1	0	3602	4207	16.8%	3602	0.0%	
	2		1802	2632	46.1%	1802	0.0%	
	4		903	1770	96.0%	902	-0.1%	
	8		607	1267	108.7%	607	0.0%	
2	1		3604	4813	33.5%	3603	-0.0%	
	2		1806	3238	79.3%	1803	-0.2%	
	4		1213	2376	95.9%	1213	0.0%	
	8		1213	1873	54.4%	1213	0.0%	
4	1		3611	6025	66.9%	3605	-0.2%	
	2		2426	4450	83.4%	2425	-0.0%	
	4		2425	3588	48.0%	2425	0.0%	
	8		2425	3085	27.2%	2425	0.0%	
8	1		4851	8449	74.2%	4849	-0.0%	
	2		4850	6874	41.7%	4849	-0.0%	
	4		4849	6012	24.0%	4849	0.0%	
	8		4849	5509	13.6%	4849	0.0%	
16	1	9699	13297	37.1%	9697	-0.0%		
	2	9698	11722	20.9%	9697	-0.0%		
	4	9697	10860	12.0%	9697	0.0%		
	8	9697	10357	6.8%	9697	0.0%		
32	1	19395	22993	18.6%	19393	-0.0%		
	2	19394	21418	10.4%	19393	-0.0%		
	4	19393	20556	6.0%	19393	0.0%		
	8	19393	20053	3.4%	19393	0.0%		
64	1	38787	42385	9.3%	38785	-0.0%		
	2	38786	40810	5.2%	38785	-0.0%		
	4	38785	39948	3.0%	38785	0.0%		
	8	38785	39445	1.7%	38785	0.0%		

RMS bound differences = 67.8% (upper) 9.9% (lower)