The Evaluation of Massively Parallel Array Architectures

Martin C. Herbordt

CMPSCI Technical Report 95-07
January, 1995

THE EVALUATION OF MASSIVELY PARALLEL ARRAY ARCHITECTURES

A Dissertation Presented

by

MARTIN C. HERBORDT

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September, 1994

Department of Computer Science

THE EVALUATION OF MASSIVELY PARALLEL ARRAY ARCHITECTURES

A Dissertation Presented

by

MARTIN C. HERBORDT

Approved as to style and content by:

Charles C. Weems, Chair

Allen R. Hanson, Member

Donald W. Towsley, Member

Wayne P. Burleson, Member

W. Richards Adrion, Department Chair
Computer Science

to the memory of my mother

Acknowledgments

This dissertation would not have been possible without the help of many people. First, I would like to thank my committee for their many helpful comments and suggestions. Specifically, Al Hanson who taught me about computer vision, Wayne Burleson who taught me about VLSI, and Don Towsley who taught me about performance evaluation. Most especially, I'd like to thank my committee chair and my advisor and mentor for my entire graduate career, Chip Weems. Besides teaching me about architecture and writing, he suggested the final form of the topic, pulled me out of many blind alleys, and his vast store of knowledge was a constant help. Many other professors at UMass also contributed to my knowledge of computer science and so helped me with this dissertation. I would especially like to thank Arny Rosenberg who not only taught me theory but more importantly how and where to apply it, and Ed Riseman who's boundless energy and optimism serves as a model for all of us.

The first level of discussion and comments is always with the fellow graduate students in one's research group. In the IUA group I would like to thank Deepak Rana who gave me my first lessons in networks and parallel processor trade-offs, Mike Scudder who played the invaluable role of idea filter during the first part of the project, and Steve Dropsho who played that role during the latter stages. The IUA group has long benefited from its close ties with the VISIONS group. I have especially learned a great deal, not only about vision, but also about how to do research from Ross Beveridge, Bruce Draper, Bob Collins, Lance Williams, Teddy Kumar, and Harpreet Sawhney. I would also like to thank Scott Anderson, Jay Corbett, and Adele Howe for many useful discussions about thesis writing and research in general.

Jim Burrill deserves a paragraph all to himself: without his initial implementation of the class libraries for the IUA and his CAAPP simulator this work would have been impossible. It was also he, in my first days as a graduate student, who showed me the finer points of parallel programming and later answered some thousands of questions about compiler and architecture internals.

I've had many office-mates who have made life here pleasant, with whom I've discussed research, and from whom I've learned much about computer science. Those I haven't already mentioned in other contexts are Alan Boulanger and Glen Weaver. There are also many others at UMASS who helped provide sanity breaks, especially Gordon Kieffer and Frank Klassner. Our good friends Rob St. Amant and Luellen Brochu played that role outside of work.

Most of all I would like to thank my wife Ellen for her constant love and support. This is very much her dissertation as well.

Abstract

THE EVALUATION OF MASSIVELY PARALLEL ARRAY ARCHITECTURES

SEPTEMBER, 1994

MARTIN C. HERBORDT, B.A., UNIVERSITY OF PENNSYLVANIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Charles C. Weems

Although massively parallel arrays have been proposed since the 1950's and built since the 1960's, they have undergone very few systematic studies and these have covered only a small fraction of the design space. The major problems limiting previous studies are: computational cost of detailed and accurate simulations; programming cost of creating a test suite that compiles to the various target architectures and runs on them with comparable efficiency; and diversity of the architectural design space, especially communication networks. These issues are addressed in the construction of ENPASSANT, an evaluation environment for massively parallel array architectures that obtains performance measures of candidate designs with respect to real program executions.

We address the computational cost problem with a novel approach to trace-based simulation. Code is run on an abstract virtual machine to generate a coarse-grained trace, which is then refined through a series of transformations (a process we call *trace compilation*) wherein greater resolution is obtained with respect to the details of the target architecture. We have found this technique to be one to two orders of magnitude faster than detailed simulation, while still retaining much of the accuracy of the model. Furthermore, abstract machine traces must be regenerated for only a small fraction of the possible architectural parameter combinations. Using virtual machine emulation and trace compilation also addresses program portability by allowing the user to code in a single language with a single compiler, regardless of the target architecture. Fairness and programmability are obtained with architecture dependent application libraries for a small set of critical functions. The diverse design space is covered by using parameterized models of the architectural components which direct ENPASSANT in the evaluation of the target machines on the basis of user specifications.

ENPASSANT has already generated significant results, including effects of varying the number of dimensions in k-ary n-cubes, trade-offs in register and cache design, and usefulness of certain ALU features. Some surprising results are that bidirectional links provide a large advantage for k-ary n-cubes (where $n = 2$) in an essential application, and that smaller rather than larger cache block sizes are favored for most applications studied.

TABLE OF CONTENTS

LIST OF TABLES

List of Figures

PART I

INTRODUCTION AND BACKGROUND

Chapter 1

Introduction

1.1 The Problem

Computer architecture research is being driven from three directions. The first is the need for ever more computing capability: many important applications including modeling, simulation, and intelligent systems (such as those that perform machine vision) require far more processing performance than can be provided by machines existing today or that will be built in the near future. As it has long been realized that we are reaching the limits of the processing capabilities of serial and small-scale parallel designs, it is clear that the only way the desired computation rates are physically possible is through massively parallel processing.

The second direction is that, in the near term, VLSI and packaging technologies continue to improve at phenomenal rates. Feature sizes have been and continue to decrease at roughly a linear rate over time. This has two consequences: switching time—which is roughly proportional to feature size—is also decreasing linearly with time; and more significantly for the purposes of this study, chip device counts are increasing *quadratically* with time. How best to use these additional devices and how to project their use as more become available are complex but essential issues. See [68] for a review of the effects of changing technology on computer architecture.

The third direction is the realization in the last decade that the approach of increasing any particular aspect of processing capability without thought of how that capability can be applied to a particular application can lead to disappointing results. A new emphasis on empirical methods has been credited with providing the framework within which reduced instruction set computers (RISC) could be developed [46]. These three driving forces together provide the impetus and mechanism for the study of massively parallel processor architecture.

The problem this dissertation addresses is how to make architectural decisions for one class of massively parallel processors—massively parallel array (MPA) computers—with respect to a subset of computationally intensive tasks we call spatially mapped applications. In the class of MPAs we include processor arrays with SIMD control and a number of processing elements (PEs) at least in the thousands. By spatially mapped applications, we mean computations that are derived from real world processes and are characterized by the fact that the spatial relationships inherent in the problem are preserved as the data are mapped to the processor array.

The solution we propose involves—at its highest level—the application of the empirical methods mentioned above. After all, in designing any complex device a systematic approach is desirable: the greater the number and the broader the range of alternatives that are examined closely, the better the chance that a good design will emerge. This is basically what Hennessy and Patterson are saying when they state that the current explosion in microprocessor performance

"... was only possible because a number of important technological advances were brought together with a much better empirical understanding of how computers are used. From this fusion has emerged a style of computer design based on empirical data, experimentation, and simulation [69]."

Our goal in making architectural decisions for MPAs is to build an environment that has the capability of accurately evaluating a large number and broad variety of design alternatives with respect to a realistic workload.

In practice there are trade-offs between the subgoals. In particular, the number of designs that can be examined must be balanced with the accuracy with which they are evaluated. At one extreme, building and testing a prototype examines only one design, although extremely accurately. The alternative to prototyping is simulation: by sacrificing some accuracy, a great deal of flexibility can be gained. Although prototyping and simulation are both essential architectural tools, simulation, because of its flexibility, generally precedes prototyping in the design process. And since relatively little simulation has been done in the domain of MPA architectures, that is the method on which we shall concentrate in this dissertation.

1.2 Issues in MPA Evaluation

The analysis of MPA architectures has not yet employed empirical techniques to examine more than a small part of the design space. Most previous architecture studies in this domain have been based on either mapping sample algorithms to architectures (e.g. [51]), requirements analysis (e.g. [139]), or feedback from benchmarks (e.g. [144]). The first two of these methods have served their purpose in making 'first cuts' at machine architectures, but now these need to be extended to yield more specific and detailed results. The third has yielded detailed results about specific designs, but has not illuminated much of the design space.

The difficulties that have prevented comprehensive studies of the kind we propose—besides the inherent complexity of such a task—are the computational intractability of the simulations and the programmability of the test suite. These are now discussed in turn, together with a preview of their solutions. But first, we define two terms: 1) *target machine* refers to the machine or design to be simulated, and 2) *host machine* refers to the machine on which the simulation is being run.

1.2.1 Computational Intractability

The simulation of MPAs at the machine instruction level requires orders of magnitude more processing than the simulation of a serial processor. This is because of the large number of processing elements (PEs) that need to be simulated and because of the size and complexity of some MPA interconnection networks. For example, running a modest sized program that takes milliseconds to run on a target machine (the CAAPP [147]) and seconds to run on a SUN SPARC-2 workstation can take hours, or even days, to simulate on that same workstation. Such long turnaround times make it impossible to examine a large number of design alternatives, even with a substantially faster host machine.

There are two keys to overcoming the computational intractability of the simulations: simulating at higher (less detailed) level, and reusing simulations for multiple designs. Both of these principles are employed in the commonly used method of trace-driven simulation. Trace-driven simulation is a two part process. In the first part, a trace is generated using a slightly less than comprehensive model of the target machine. We call this a *detailed* simulation, as opposed to a *complete* simulation where all components would be modeled. For example, though the instruction and register architectures are specified in a detailed simulation, typically the cache and pipeline architecture are not. This brings us to the second part of the process: the trace can now be used any number of times to drive simulators of those components that were not initially modeled. Trace-driven simulation works because the design of the components being modeled does not affect the program execution that drives the generation of the initial trace.

Trace-driven simulation, however, does not help us address the problem of computational intractability (nor of programmability and portability). The initial detailed simulation is still too costly. In fact, for target machines with simple PE ALUs the detailed and the complete simulation are identical. Instead, our approach is to run the initial simulation and trace generation at an even higher level—on what we call the MPA *virtual machine*. The MPA virtual machine is the minimum configuration necessary to run a program written in a generic MPA language. The trace generated from the virtual machine *emulation* is then passed through a series of transformations with respect to the parameters and features of the target architecture until a trace emerges that closely resembles the trace that would have been generated by a detailed simulation of that target machine. We call this process of trace reconstruction *trace compilation*.

Together, virtual machine emulation and trace compilation are typically 30 to 50 times faster than detailed simulation. Also, traces need to be generated far less frequently. See Figure 1.1 for block diagrams of the different methods described in this subsection.

1.2.2 Programmability and Portability

There are two portability problems. The first is that there does not exist a high-level language that is supported for more than a small number of MPAs. The second is that MPAs are such a broad architectural class that porting code among target architectures (or potential designs) sometimes requires that functions be recoded to use different algorithms. We now describe these problems.

1. It is essential that programs written for our simulation system be portable in the sense that they are compilable and executable no matter what target machine we would like to examine. This is where the virtual machine methodology yields another benefit: as long as the input program runs on the virtual machine emulator, the particulars of the target machine are irrelevant.

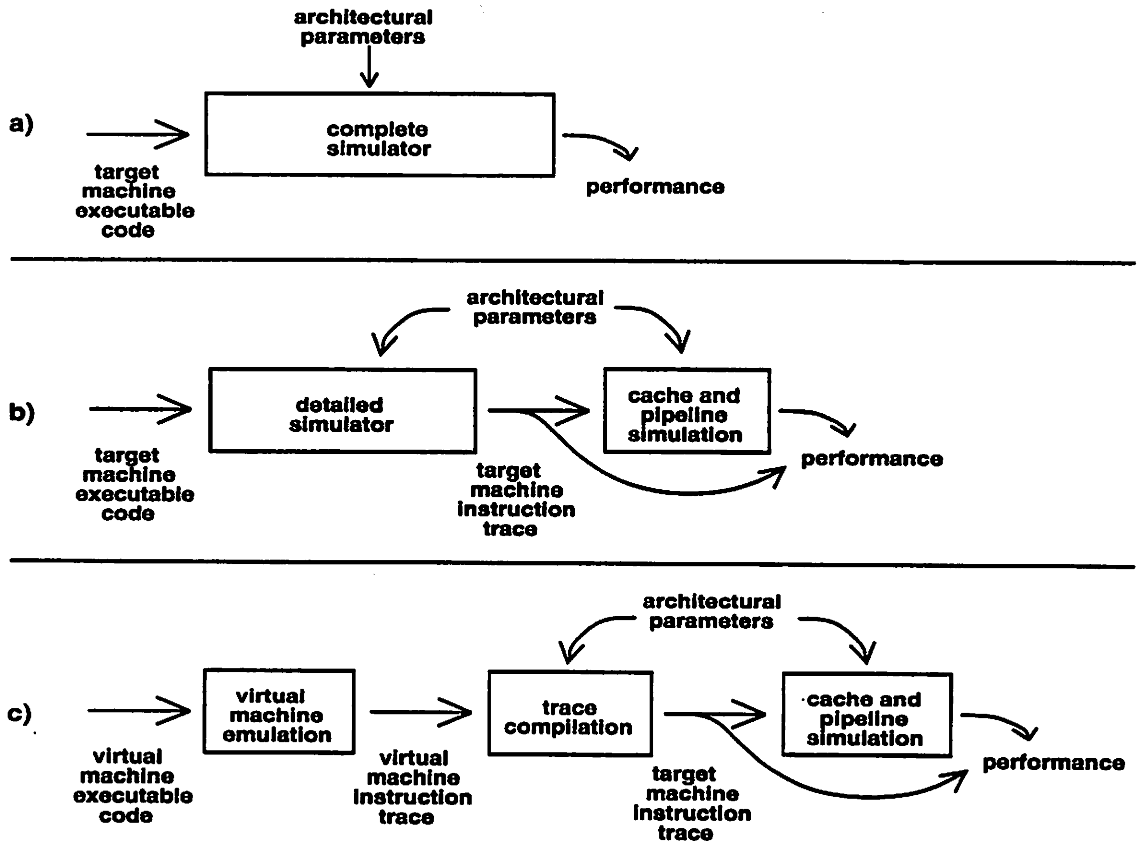

a)
architectural parameters
complete simulator
target machine executable code
performance
b)
architectural parameters
detailed simulator
cache and pipeline simulation
target machine executable code
target machine instruction trace
performance
c)
architectural parameters
virtual machine emulation
trace compilation
cache and pipeline simulation
virtual machine executable code
virtual machine instruction trace
target machine instruction trace
performance

Figure 1.1. Block diagrams of three possible methods of evaluating a class of architectures: a) complete simulation, b) trace-driven simulation, and c) virtual machine emulation and trace compilation.

There is, however, an important issue that must be addressed before this can happen: certain language constructs that are an essential part of the programmer's model of the MPA virtual machine are not supported directly in hardware by all target machines. In this situation, those language constructs must be emulated using features that *are* available on the target architecture.

2. Porting code among MPA architectures (or potential designs) sometimes requires that functions be recoded to use different algorithms. Otherwise the appropriate features will not be used properly and the results will be skewed.[1] This problem can be viewed as balancing programmability with fairness and accuracy: either the benchmark is task oriented and requires a coding effort for each significantly different platform, or it is source code oriented and maps unevenly to different designs. Vision architecture benchmarks, for example,

[1]This version of the portability problem can usually be handled in serial architecture studies by having a high quality compiler available for each target architecture. This is not the case for massively parallel architectures for reasons that will be discussed later.

have leaned towards being task oriented [120, 124, 148] and have therefore depended on independent efforts by each architecture's advocates to code the test suite. This has again limited the performance measurements to a few specific machines.

To maintain fairness while still allowing the search of a significant part of the design space, we use a combination of task oriented test suite specification and architecture dependent object libraries. The basic idea is to provide different versions of particular sub-tasks to those architectures that require them, but to do this only when they are needed.

See Figure 1.2 for block diagrams of how the emulation and task libraries fit in the system.

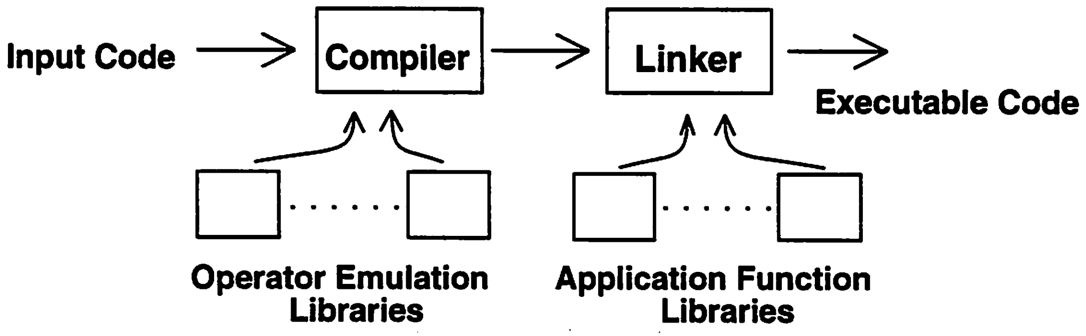


Figure 1.2. Portability requires libraries to emulate unavailable hardware and to provide different versions of algorithms as called for by variations in hardware support.

1.3 The Approach

The overall problem addressed in this dissertation is how to create a mechanism with which to investigate MPA architectures by simulating instances of target machines with respect to real program executions. The primary result is an evaluation environment for MPA architectures that is accurate, flexible, efficient, fair, and programmable. We call it ENPASSANT (ENvironment for PArallel System Simulation ANalysis Tools).

To get from the overall problem to the primary result involves solving many subproblems besides those mentioned in the previous section. These key issues and our approach in addressing them are briefly discussed in this section.

1.3.1 The MPA Architecture Space

The architecture space, or the set of features and parameters that form the domain of MPA architectures can be partitioned into those components which are common to all machines in the class, and those that are optional.

We assume that all target architectures in our study are composed of a controller and an array of PEs running in SIMD mode. The controller broadcasts instructions and global data to the array and has the capability of reading global status in the form of a global OR. The PEs in the array are simple, having no micro-sequencer or instruction autonomy other than the choice of participating or not participating in a particular instruction. The PEs are interconnected via a mesh network, or a network that can efficiently emulate mesh connections.

The optional features include dedicated routing networks that perform broadcast, permutation, scan, and reduction operations; floating point support either in the form of co-processors, or local hardware enhancements such as barrel shifters; other PE features such as multipliers; PE cache memory; and many others. Many features have parameters that can be varied: examples are the number of registers, the ALU width, and the dimensionality of the router network.

1.3.2 Selecting a Test Suite

A problem with current MPA architecture research is that the benchmark test suites may not have accurately reflected the workload of spatially mapped computations. They have often been restricted to relatively small computations and a set of well-known—but not necessarily representative—algorithms. Recent efforts have gone some way in changing this (see e.g. [148]).

We address the problem of proper workload selection by including applications in our test suite that are in current use; that have more than trivial size, ranging from a few hundred to a few thousand lines of code; and that, as much as possible, span the space of the types of computations likely to be encountered in practice.

1.3.3 Creating a Generic MPA Language

As we mentioned above, there does not currently exist a language supported by even a small number of MPA target architectures. A minimum requirement of such a language is that it give the programmer access to the features available on the particular target machine for which the code is intended. Since our MPA language is meant for not just one target architecture, but rather for the entire class, the programmer's model must contain access to the union of the features available on MPAs.

The basis for our 'generic' MPA language is ICL, a parallel class library extension to C++ [39]. First developed for the IUA [147], we have extended it to support the aforementioned union of MPA features. In other words, the ICL programmer's model is a generic MPA; what we have already referred to as the MPA virtual machine model.

1.3.4 Emulating Operators for Optional Features

A programmer will often select a language construct for a particular task on the basis of convenience, rather than on performance. And in any case, once a program has been written in ICL, we want to be able to use it to evaluate any MPA target machine. That means that a construct C that was included in ICL to give the programmer access to hardware feature F that is available on some, but not all, MPAs, must also be executable on target machines that do *not* have F. To do this requires that C be emulated with constructs that are available on the target machine. Examples include emulating permutation routing on MPAs with no dedicated router network by using nearest-neighbor connections, and emulating region broadcast on machines with no broadcast network with a network that efficiently executes scan operations.

1.3.5 Application Library Functions

As the previous subsection illustrates, porting a code directly from one processor to another means that available hardware features are sometimes used suboptimally. To prevent this from skewing results, at least in the most serious cases, we have constructed an application function library. Those sub-tasks within the benchmark programs whose choice of algorithm is hardware dependent have separate versions provided. The appropriate version is automatically selected according to the target architecture. Examples of these tasks are labeling connected components and finding convex hulls.

1.3.6 MPA Virtual Machine Emulation and Trace Generation

The method we use to emulate the MPA virtual machine is to compile the ICL test suite program directly onto the host machine and then run it. Directives are embedded in the host machine executable code to generate a trace record for every virtual machine instruction that is emulated.

1.3.7 Trace Compilation

Virtual machine code is in some ways analogous to code generated by the front end (the machine independent part) of a compiler. The trace is necessarily also machine independent. The method we use to generate a target machine instruction trace from the virtual machine trace is analogous to running code in an intermediate representation through the back end (machine dependent part) of that same compiler. The functions performed include register allocation and assignment, target machine instruction generation, and peephole optimization.

1.3.8 Evaluating MPA Components

The purpose of ENPASSANT is to evaluate MPA designs. However, much efficiency can be gained if all design components are not specified or evaluated simultaneously. For convenience, we partition the space of MPA components being evaluated into four distinct sets: the first three are the router networks, the registers, and the cache; the fourth consists of the remaining features including the PE internals. See Figure 1.3 for a block diagram of where the specification and computation occurs for the evaluation of the different components.

We now give some examples.

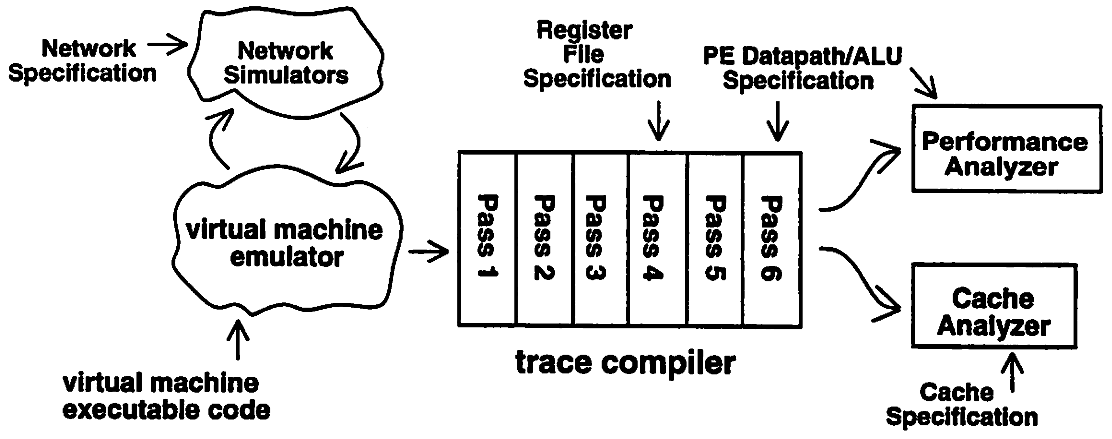


Figure 1.3. The different components of the architectural design space are specified and evaluated in various parts of ENPASSANT.

- The performances of dedicated router networks are sometimes highly data dependent. They must therefore be at least partially simulated during virtual machine emulation.
- The register file architecture affects the generation of the memory reference trace which takes place during pass 4 of the trace compiler. It must therefore be specified at that point. The effect of the register file specification can be measured further downstream, however.
- The cache, which depends on the virtual memory address trace, can be evaluated independently from the rest of the architecture. This situation is identical to that in trace-driven simulation.
- All other architectural components, including the PE datapath and ALU, the array to controller feedback, and the nearest-neighbor and broadcast communication are specified during the last phase of trace compilation. This is where virtual machine instructions are expanded into target machine instructions. The performance analyzer uses the resulting target machine instruction trace to obtain the performance.

1.3.9 Virtual Processor Support

A general rule in running codes on MPAs is that the number of data elements greatly exceeds the number of PEs. The way this situation is usually handled is to map the data for one element to each *virtual* PE, and then have the physical PEs emulate as many virtual PEs as necessary. ENPASSANT supports two types of PE to virtual PE mapping: block and cyclic.

Since some of the communication instruction emulations depend on the ratio of virtual to physical PEs (the *virtualization factor*), the processor array size (number of physical PEs) and the mapping style must be specified at run time. The rest of the virtual PE emulation code is generated during trace compilation.

1.3.10 Traversing the Evaluation Space

There are at least two uses for ENPASSANT

1. to be a tool in examining the basic issues in MPA design, such as processor granularity and the complexity of the inter-PE communication network, and

2. to study existing designs, that is, benchmarking and examining the effects of future parameter and feature changes.

These two uses obviously overlap: for example, existing designs are naturally good starting points for searching the MPA architecture space.

Three components of the MPA design space are largely independent from one another. These are the memory hierarchy; the dedicated communication network, if there is one; and the rest of the array, consisting primarily of the PE ALU and datapath, the array to controller feedback, and the nearest neighbor connections. As in all system design, all else being equal, we seek to remove bottlenecks; or conversely, to achieve system balance.

1.4 Outline of Results Achieved, Contributions

We now describe the contributions of this work and their significance. We begin with contributions that arise from building the evaluation system.

- ENPASSANT: an evaluation environment for simulating MPA architectures with respect to real program executions. Especially noteworthy is that the environment is flexible, efficient, fair, and accurate. The significance is that ENPASSANT will enable architects to explore the MPA design space in much more detail than was previously possible. No current system has the flexibility and efficiency necessary to provide this function.

- The virtual machine and trace compilation methodology. Although other evaluation methods improve performance by omitting detail and reusing computation, the idea of a trace compiler that reconstructs virtual PE emulation, register allocation, optimization, and target machine code generation is completely original. The trace compiler has two performance benefits. The first is that it allows us to simulate MPAs at a high level (virtual machine emulation); this gives us more than an order of magnitude speed-up in evaluation time over detailed simulation. The other benefit is that traces need to be generated far less frequently; only when there are changes in the number PEs and the type of communication network. This is in contrast to other methods where the trace would also need to be regenerated for changes in register file size and target machine instruction set. The significance is that our method gives us the efficiency required to systematically explore the MPA design space that is not offered with other methods. Also, the trace compilation method may be useful in some domains where trace-driven simulation is now used because of the reduction in the frequency with which traces need to be generated.

- Selective recoding. By using application function libraries we recode those and only those parts of the applications that require it. A modest programming effort was found to be adequate to enable fair comparisons for the architecture and application domains in this study. The significance is that it seems likely that this result will extend to other application areas and possibly to other architecture domains. Although function libraries (graphics, numerical methods) are ubiquitous, libraries that contain a number of functions—each of which is implemented with multiple algorithms whose choice depends on the architecture—are uncommon. However, work has been done elsewhere in the development of programming methods that allow code to be ported with minimal recoding and little loss in performance [8].

- Portable language. By adding a number of constructs and an emulation library to ICL, we have created a language that allows us to use the same program during the evaluation any MPA target architecture without recoding.[2] Although the idea of emulating optional hardware is an old one (e.g. floating point emulation versus floating point coprocessor), it has not been applied to the emulation of interPE communication instructions on MPAs. In fact, earlier implementation of the emulations would have been difficult as sufficiently efficient algorithms for several of them were only recently developed [70]. The significance is that ICL now has a greater potential to form the basis for a portable language among machines in the MPA class for the class of spatially mapped applications.

- The benchmark test suite. We have assembled a suite of non-trivial programs having diverse arithmetic, communication, and memory requirements. The significance is that together they provides a more complete test bed than was previously available, say, by using the DARPA IU benchmark (see [148]) alone.

The other major contributions of this work are the results that have been derived so far using ENPASSANT. Their significance is that they provide recommendations for the next generation MPA processors. Selected results are as follows.

- **Datapath.** When the performance of the test suite programs is measured as a function of ALU width, most of the gain occurs when the ALU width is increased from 1 to 8, and especially, very little performance is gained when the ALU is increased from 16 to 32.

- **Register file.** When the cost of memory references is measured as a function of the number of registers, distinct working sets were found for all programs. This indicates a recommended register set size for future MPAs.

- **PE Cache.** Associativity has a major effect on PE cache performance: direct mapped cache must often be 2 to 4 times larger than fully associative to achieve the same performance.

[2]Though not necessarily fairly. See previous point.

Also, PE memory references do not have much locality: thus very small block sizes are preferred.

- **Packet switched (k-ary n-cube) networks.** In an essential application, performance improved little when the dimensionality was increased beyond three (and the bandwidth held constant). Also, bidirectional links provide a large advantage when n = 2.

1.5 Organization of the Disseration

The rest of the dissertation is organized as follows.

- In Chapter 2 we review the common techniques used in architectural research and especially in architectural evaluation. We find that none of the existing methods meet our requirements.
- In Chapter 3 we describe in detail the MPA design space and the components that will be examined and evaluated.
- In Chapter 4 we review the application space—the types of programs that are typically run on MPAs by end users. From these programs we derive a set of common features. We then present a test suite that spans this feature space.
- In Chapter 5 we give a high-level overview of ENPASSANT.
- In Chapter 6 we describe how ENPASSANT deals with the programmability and portability issues. We discuss the selection of the MPA virtual machine programmer's model and the contents of the application function and operator emulation libraries.
- In Chapter 7 we give the details of the virtual machine and trace compilation methodology. Included are discussions of the issues involved, especially validation and performance.
- In Chapter 8 we begin the descriptions of how we evaluate the components of the MPA design space with a discussion of the datapath. Also presented are an overview of the datapaths and ALUs currently in use, the parameterized datapath model we use to abstract that space, and some case studies of the effects of varying parameters on instruction and program execution times.
- In Chapter 9 we continue describing component evaluation with register file and cache design. The organization is similar to that in chapter 8.
- In Chapter 10 we examine communication networks. Again, the current design space is presented together with the models we use to abstract those spaces and followed by the case studies.

- In Chapter 11 we present some more sample results that have been obtained using ENPASSANT.

- In Chapter 12 we present our conclusions and describe areas of future research.

Chapter 2

Massively Parallel Array Evaluation: Issues and Review

Computer architecture research can be viewed in some respects as a search problem. As with most search problems, there are two aspects: creating the search space and traversing it. In this view, creating the architectural search space can be seen as taxonomizing existing designs and incorporating new design alternatives, while traversing the space can be seen as evaluating them.

There are two primary, often inter-related, sources for new architectural alternatives: the application of new technologies to machine design (e.g. integrated circuits), and the invention of new components (e.g. various routing networks). Although there is much active research in these areas, this dissertation will concentrate on the second part of the architecture research problem: how to traverse the space defined by architectural design alternatives.

Each step of a search space traversal consists of two parts: evaluating a point in the space of potential architectures and deciding which of the very large number of alternatives to try out next. The purpose of this chapter is to show that for an important class of computer architectures—massively parallel arrays—no adequate mechanism currently exists to perform this function. Later we will demonstrate that, as a consequence, relatively little of the massively parallel array architecture search space has been traversed.

The rest of this chapter is organized as follows. We begin with a brief description of general issues in architectural evaluation, followed by a presentation of commonly used evaluation techniques. After that comes a discussion of the issues that arise in applying these techniques to massively parallel arrays and of the ways these issues have or have not been dealt with in previous evaluation systems. This chapter ends with a brief outline of how these issues are dealt with in the rest of this dissertation.

2.1 General Issues in Architectural Evaluation

Evaluation implies a metric in which the appraised value of the evaluated object can be expressed. In the case of computers, the most important metrics are performance and various kinds of cost. We deal almost exclusively with performance. *Computer* evaluation further implies a workload that runs on the computer to be evaluated; in other words, target machines must be evaluated with respect to a particular workload.

The ideal way to evaluate the potential performance of a processor design is to build it and then to use it to run exactly the workload for which the processor is intended. Then the design's performance can be determined precisely by simply measuring the execution times of the jobs in the workload. This approach is obviously impractical, however, as building systems is very expensive.

A number of methods have been developed that are used to approximate this ideal evaluation strategy, but they trade off the accuracy and/or confidence obtained from the trial against the cost of the trial (often in terms of execution time). We briefly discuss two issues involved in this trade-off: approximating the workload, and constructing a platform on which to run the approximated workload. Some of this material is derived from the fine survey by Heidelberger and Lavenberg [67].

2.1.1 Workload Approximation

Machine performance is often highly workload dependent, so characterizing the workload as precisely as possible is central to accurate evaluation. One extreme is to run the entire workload, but this is often much too time consuming even if it is known *a priori*. Instead, the workload is usually approximated. If the workload can be characterized by probability distributions of key characteristics, then mathematical models can be used. This is rarely the case with complete computers, however. More commonly, a set of tasks that approximates the workload but which can be executed more quickly—a *test suite*—is used.

The closer the test suite approximates the true workload, the more accurate the evaluation. The trade-off is usually against the amount of time required to obtain the results. Obviously in the extreme case, the test suite equals the workload.

Another issue is matching the test suite against the workload. There exist many public domain test suites, some of which will be discussed later. The test suite tasks can be real or synthetic. In general, however, it is up to the end user to determine which particular test suite tasks come the closest to representing his or her own workload.

2.1.2 What to Run the Workload On

If the workload can be characterized with mathematical models, then analytical techniques can be used to approximate the performance. Otherwise a platform must be constructed on which to run the test suite. The two common alternatives are prototyping and simulation.

The advantage of using prototypes, especially those with built in instrumentation (see e.g.[42]), is that they guarantee accuracy with very fast turn-around. However, it is readily apparent that building prototypes is expensive and time consuming and also that, once built, a prototype is difficult to modify.

A cost-effective alternative, especially early in the evaluation process, is software simulation. The biggest benefits of simulation are that it is much easier to construct a simulator than it is a prototype, and that simulators can be parameterized. That is, a simulator can be constructed with built-in flexibility so that it can emulate a number of different prototypes, or even an entire class of architectures. Another benefit of simulation is that there is usually the opportunity to trade off the accuracy of the evaluation for the time spent adding detail to the simulator and the amount of time spent running the test programs.

The drawback is that executing a complete simulation, especially for a parallel machine, is extremely time consuming [12]. Also, a model may be hard to validate if the target machine is still early in the design stage [109]. There is also the issue that a machine can never be fully characterized until it is constructed, to quote Yale Patt: "many problems attending a new computer architecture or implementation do not surface until you build and perform experiments on real hardware [115]."

The choice, however, does not have to be only between spending resources building prototypes versus spending time simulating them. There are also hybrid techniques where host machines can be used to approximate target machines, and where information from some parts of the evaluation process can be reused. These are discussed in detail below.

2.2 Techniques Used in Architectural Evaluation

In this section the primary techniques in use by the computer architecture community are discussed. These mostly involve observing the behavior of real and synthetic systems, i.e. experimental rather than analytical techniques. As the latter are generally limited by the fact that real machines are largely inelegant and therefore not easily modeled [115], they will not be discussed further.

2.2.1 Design From Requirements

When the general application area for which the target architecture will be used is known, but the specifics are not, ***design from requirements*** is often an effective tool. For example, it might be possible to determine the rate of computation, or that a flexible instruction set, or a certain amount of memory is needed. These requirements can then form the basis for the rest of the design. Here are a few examples.

While examining the pattern recognition problem Unger made two fundamental observations. The first was that conventional computers can only operate on a small amount of information at a time; the second was that it would be advantageous to build a machine that could operate "directly on information in planar form [the way it is naturally arranged] without scanning or using other techniques for transforming the problem into some other domain [142]." The product of these observations was one of the first parallel computer designs.

We move forward thirty years to the next example. Tsotsos has examined the problem of immediate vision [139], i.e. visual perception that is not influenced by higher order considerations and does not involve active intelligent examination [101], and found the following novel technique for generating an architecture. The starting point is a computational analysis of immediate vision which concludes that the straightforward solutions are intractable. However, since the human brain is quite capable of performing immediate vision, it follows that there must exist a series of global optimizations that make the problem tractable. Tsotsos next develops a set of biologically plausible optimizations which are then used to form the set of requirements for a parallel processor to perform pre-attentive vision processing. The resulting processor is a set of receptive field assemblies or columns of arrays connected to their own relevant retinoptic (or *spatially mapped* [142]) elements.

Weems examines the field of vision research as a whole and compiles lists of operators, functions, and data representations either currently in use, or generally agreed to be essential to some aspect of the vision problem [145]. Using a classification scheme commonly used by the vision community (see e.g. [32, 123]), the lists are divided into low-, intermediate-, and high-level sub-lists. The items in each sub-list are examined and found to share certain important computational characteristics. The sub-lists thus form the basis from which sets of requirements for three distinct processors are created. Overall requirements of the vision tasks are then used to determine that these processors must be tightly coupled to provide the rapid feedback that must take place among the computational components. The resulting design is described in detail in [147].

2.2.2 Application Specific Processors

Application specific processors are popular in areas where general purpose processors do not yield satisfactory performance for some set of end users and where those users are willing to pay extra for additional processing capability. The most successful application specific processors are those developed for applications that have high demand and for which large performance speed-ups can be achieved at modest cost. Examples are floating point coprocessors, signal processors, graphics (co)processors, and processors that perform certain image processing tasks such as convolutions and Fourier transforms.

We mention application specific processors here because there is some overlap between that domain and the object of our study, massively parallel arrays for spatially mapped applications. For example, a popular application area for research in application specific processors is the subset of spatially mapped computations referred to as image processing: that is, operation where both the input and output are images or their transformations. Also, some popular application specific processors are massively parallel arrays.

However, because the domain of application specific processors is characterized by the fact that the application of interest consists of a modest number of well-defined tasks, the methods used in the development of application specific processors are quite different than the ones necessary for broader studies such as ours. The usual technique is the analytical characterization of the operations in the domain of interest with respect to the architectural alternatives. We mention some of the more important work that uses this technique and which overlaps our domain.

- Reeves studies MPA alternatives for image processing [121] and finds that the key functions that must be implemented efficiently are boolean operations within PEs, nearest neighbor moves, and recursive nearest neighbor moves. A binary array processor with an appropriate instruction set is proposed.

- Komen also studies the domain of image processing [88] and finds that non-linear recursive neighborhood operations are a key component in the efficient implementation of several tasks. The performance of these operations is evaluated with respect to several groups of architectures and the conclusion is reached that linear processor arrays offer the best price-performance ratio.

- Jonker expands the study to morphological image processing [82] and proposes an architecture that combines the linear array and pipeline processing models.

- Sunwoo and Aggarwal concentrate on the image processing sub-tasks that are based on window operations [134]. They propose an MPA with certain optimizations, the most significant of which speed up nearest neighbor communication.

The drawback of this technique is that many applications, including the ones in which we are interested, are much too complex to be characterized analytically.

2.2.3 Code Profiling

Code profiling is the process of determining the distribution of the individual instructions in an instruction set that were executed during a particular program execution. In systems where instruction execution times are deterministic, performance can be found by computing a weighted sum using the instruction histogram and the known execution times. Therefore, in cases where representative application codes are available and where the order of the instructions in the execution stream does not affect performance, profiling can be a very effective evaluation technique. However, since profiling fails to capture the effects of temporal instruction distribution, it is less effective in evaluating systems where the performance of the memory hierarchy and datapath pipeline are application dependent.

Profiling is most useful for making coarse determinations, such as whether special purpose add-ons are likely to be cost-effective. For example, if the decision is to be made between adding a floating point co-processor or not, knowing the frequency of floating point operations in the critical codes is likely to be sufficient to make that decision.

2.2.4 Evaluation Using Representative Sub-Problems

Many architectural studies fall into a category that could be called 'evaluation with respect to representative sub-problems.' The goal is to demonstrate the performance of the design in a certain application domain by coding some standard algorithms, running them, and determining the performance. Some machines and proposed machines that have been evaluated in this way are the Connection Machine [140, 141, 98], the Polymorphic Torus [93], the Mesh with Multiple Broadcast [118], and the Aspex ASP [89]. This technique is most useful in providing proof of a basic concept.

2.2.5 Benchmarking

A more systematic approach to evaluating existing machines and machine designs is by standardizing the above process through organized benchmarks. According to Duff [56],

> "The purpose of benchmarking a computer architecture is to establish a figure of merit for the architecture with a view to justifying a particular design strategy."

Benchmarking is a complex and evolving area as is indicated by the number of test suites that have been and continue to be developed. Some of the issues involved in test suite selection will be discussed in the rest of this subsection.

A basic principle of benchmarking is that the more closely the test suite resembles the prospective workload, the more accurate the evaluation will be. However, since the purpose of benchmarks is to approximate as many workloads as possible, a balance between ease of use and usefulness must be achieved. On the one hand, a test suite that is very time consuming to run or which produces results that are difficult to analyze will be worthless since few users will want to use them. On the other hand, a simple benchmark can be easy to implement, but produce results that have little value because only a small subset of the potential workload is used to exercise the target machines.

Another issue that must be addressed in the benchmark test suite is that many systems do not exhibit their true behavior until after a significant 'warm-up' period. For example, certain small benchmarks do not exercise the memory hierarchy at all; with the growing size of cache memories, the hit ratios are approaching 100% [149]. An emerging consensus has thus been that many designs will not be exercised correctly with anything less realistic than real program executions.

Many other difficulties with benchmarks are well known; two of the most important mentioned by Duff are [56]:

- **Task Definition.** Deciding on what tasks are essential, their relative importance, and which can be left out; in other words, finding the most cost-effective representation of the workload.

- **Algorithm Definition.** Specifying the precise algorithm to be used can force a poor method onto a good architecture. However, leaving the algorithm up to the implementer can make the benchmark into a programming contest.

As a result of the second difficulty, two distinct approaches to benchmarking have been developed: we shall refer to them as *code oriented* and *task oriented.*

The *Systems Performance Evaluation Cooperative [SPEC]* effort is an example of a code oriented benchmark. It consists of ten program codes including a compiler, an interpreter, and various simulators and linear algebra routines. In order to compare architectures using a code oriented benchmark, the only requirement is that a compiler exists to run the program codes on each target machine. Once executable versions have been determined, performance is obtained by running the codes and measuring the elapsed time. Several studies using the SPEC approach have appeared [45, 66, 23].

The disadvantage of code oriented benchmarks, especially for massively parallel processors, is that the performance of a given task on a given processor is often highly algorithm dependent. More significantly, the best performance for a particular task is often affected by different algorithms on different processors, even within the same class of architectures. If the benchmark contains such a task, then the evaluation can be seriously skewed.

This problem is dealt with in task oriented benchmarks by specifying the *task* to be run, rather than the program code. Examples of task oriented benchmarks are the ARPA IU II [148] and the NAS benchmarks [13]. The problems with task oriented benchmarks are derived from the fact that they may require separate coding efforts for each target architecture. One consequence is that the evaluation can become a contest in programmer skill. Another is that creating a version of benchmark to run on a particular platform sometimes requires programmer years, a cost that is often unacceptable.

Another problem with current test suites is that even the SPEC and similar benchmarks are incomplete in their modeling of prospective workloads of general purpose processors. Some applications are completely ignored; the negative effect of ignoring PROLOG support, to give just one example, is described in [76]. Also generally ignored is interaction between application program and operating system, a crucial factor in multitasking and multiprocessing environments. Anderson et al. describe the negative effect of RISC processor designs on interprocess communication, virtual memory, and thread management [10]. Among their conclusions are that the increases in size of system state, cost of system calls and interrupt handling, and byte copying are particularly to blame. These effects could not be discovered by running test suites comprised entirely of application codes under controlled conditions.

We mention one more benchmarking pitfall: trying to summarize performance with a single number. Although this practice is ridiculous for any purpose besides the grossest order-of-magnitude comparisons (or for extremely special purpose tasks, say evaluating FFT processors), it is nonetheless widespread. At the very least, if this method is used, certain practices should be followed: e.g. using the geometric [61] or the harmonic mean [129] rather than the arithmetic mean to average results.

2.2.6 Simulation

There remains the question of what to run the benchmark test suite on. Clearly, running benchmarks on existing processors is critical in refining designs. However, we are also concerned with a largely unexplored architectural space. Because of the aforementioned difficulty with building prototypes, simulating potential designs is critical. This section describes several popular simulation methods.

2.2.6.1 Complete Simulation

The most obvious simulation method is to simulate an entire processor. The advantage of complete simulation is that, with care, accurate results for prototypes can be generated. The drawback is the large amount of time that complete simulation takes even on the most powerful host processors.

There are numerous levels of detail at which this simulation can possibly take place. One is to simulate at the gate level. While this may be useful to verify the correctness of a circuit design, it is too time consuming to be of much use in anything more than determining that instructions are being executed properly. A more useful level of detail is to simulate at the machine instruction level.

Examples of such systems are the S-1 MkIIa multiprocessor simulator [12], the system used by Lilja to evaluate cache-coherence strategies for shared memory multiprocessors [97], and the simulator for the CAAPP [146]. However, even at the instruction level, the time required to run a simulation frequently needs to be measured in days. For example, common applications that would take milli-seconds to run on the CAAPP and minutes to run on the Sun Sparc-2 processor can take from hours to days to run on the instruction-level CAAPP simulator.

The consequences are obvious. Recall that architecture research is a search problem. The quality of the search, and by extension the quality of the end-result, is directly related to the number of points in the search space that have been examined. If only a few points can be examined per week, then the pace of search space traversal may be unacceptably slow. However, complete simulators are still extremely useful for 'pre-arrival' code development, benchmarking detailed designs, evaluating *particular* design changes, and validating coarser simulation methods.

2.2.6.2 Trace-driven Simulation

The problem with complete simulation is the necessity of running a time-consuming process for each architecture/application data point. Trace-driven simulation improves on this technique by allowing much of the work expended during the simulation to be reused.

Once the instruction set architecture has been determined, the components with instruction order or memory location dependent performance can be evaluated flexibly and accurately by running instruction sequences (or *traces*) through a simulator that models those components. Instruction and/or memory reference traces are obtained either from a detailed simulation or from an instrumented prototype. The trace is then used to drive simulators for the datapath pipeline or memory hierarchy.

Trace-driven simulation has long been one of the essential tools of the computer architect, being especially useful for evaluating memory hierarchy designs and replacement policies. Belady was one of the first users [20]. Smith has written a definitive survey on caching in which several new results were obtained on the basis of extensive trace-driven simulation [128]. Trace-driven simulation also has other uses: Hsu and Banerjee used this technique to evaluate a hypercube multicomputer with respect to communication behavior and CPU utilization [77]. Lang et al. [90] use trace-driven simulation to model pipelines. Many other papers have and continue to appear on this subject and can be found in the proceedings of the various computer architecture symposia [1, 2, 3].

One problem with trace-driven simulation is that the trace must still be generated through an initial simulation or prototype execution. That is, if no prototype is available, then time-consuming complete simulations must still be executed, albeit much less frequently than when no trace is generated. If a prototype *is* available, then there is the further problem that it is only possible to store at most a few seconds of target machine execution time. A problem in collecting traces for multi-processors is that this process in inherently non-deterministic: the time at which an event is recorded is not the time at which the event occurred [64].

2.2.6.3 Execution-Driven Simulation

The major problems with trace-driven simulation are the amount of storage required to save the traces and/or the execution time of the initial trace generation. A method that addresses both these problems is *execution-driven simulation.* The key idea is to actually run the test program on the simulation host machine and to calculate the target machine performance as the test program is being executed.

Execution-driven simulation is best explained through an example. This description is of the Rice Parallel Processing Testbed [48].

Assume a host processor H, a target processor T and a test program P. P is compiled with respect to T. The basic blocks of this machine code are examined and timing information obtained for execution of basic blocks of P on T. P is then compiled with respect to H. The basic blocks of the P_H code are matched to the basic blocks of the P_T code. Code fragments are then inserted before each basic block in the P_H code. These fragments, when executed, write the timing information obtained from the T compilation/evaluation to a buffer. Execution-driven simulation thus behaves similarly to a profiling tool, except that the information recorded is not of the program code running on the host; rather it records the information of what the program execution would have been on the target.

If H and T are both uniprocessors with similar organizations, this process is straight-forward, although there are several problems that must be solved. These include matching basic blocks and obtaining timing information of the target processor. For the case where H and T are both multiprocessors, or for the common case where H is a uniprocessor and T is a multiprocessor, execution-driven simulation is more complex. In the latter case, the following procedure is used.

The test program P is comprised of a series of threads $(P_0, \ldots, P_n)$ that are designated to run on the target processing elements $T_0, \ldots, T_m$. The host processor H runs each thread P_i in turn, recording the P_T timing information, until an interprocessor communication or a synchronization operation is reached. These operations are simulated using a special communication simulator and the timing information recorded. The host then processes the next thread.

Covington et al. have shown that execution-driven simulation can work well for multiprocessors with overheads less than a factor of 20 typical. They have also verified their simulations to within 10% for several machines [47]. Several other systems have also been developed that use execution driven simulation [151, 55, 133, 34, 31, 122, 53].

One difficulty of execution-driven simulation is that for processors where H does not resemble the processing elements of T, the basic block matching process can be difficult. Another difficulty is that code must be recompiled and reexecuted for every change in the design of the target architecture.

Execution-driven simulation is used primarily when the host machine resembles either the target machine, or at least the processing elements of the target.

2.3 MPA Evaluation: Issues and Previous Work

In the preceding section, several evaluation techniques were presented. In general, the more sophisticated ones (trace- and execution-driven simulation) have only been used in the domains of uniprocessors and multiprocessors and not for MPAs. In this subsection, some of the reasons for this lack of usage will be discussed.

2.3.1 Difficulties in MPA Evaluation

Some of the attributes of MPAs that make them difficult to evaluate are as follows:

- MPAs are big. They typically contain 10's to 100's of thousands of PEs.
- Architectures in the MPA class are composed of a wide variety of components, especially routing networks.
- PEs are very different from SISD processors.
- MPAs are not in common use.

The consequences are as follows.

- **Complexity of MPAs.** Because MPAs typically contain many thousands of PEs, complete simulation is especially expensive. As was mentioned previously, running a modest test program on the CAAPP simulator can take several hours, if not days.
- **Inadequate Language Support Across the Class.** Because there is a wide variation in architecture within the MPA class, a language designed for the entire class necessarily contains constructs that are supported directly in hardware by some—but not all—processors in the class. If a construct is not supported directly in hardware, it must then be emulated using the hardware that is available. For example, an essential operation in any language supporting MPAs is routing arbitrary permutations. Since the MasPar MP1 and

the Thinking Machines CM2 have network support for this operation, the corresponding language construct can be mapped directly to those processors. However, MPAs not having such support (MPP, DAP, CAAPP, etc.) require emulation using communication support that is available (nearest-neighbor communication, coterie network, broadcast buses, etc.). Such emulation libraries have not been available.

- **Architecture Dependent Algorithm Selection.** Because MPAs have so many PEs, the variation in some features, such as routing networks, is much more critical than it is in smaller scale parallel processors. For example, there is a very large variation in inter-processor communication times among architectures in the class. A permutation operation for a 16K PE MPA can vary from 50 microseconds to 10 milliseconds. More critically, the function mapping the routing pattern onto time required to execute the communication is highly network dependent. As a consequence, the choice of algorithm for any particular task is likely to vary depending on the network.

- **Non-standard PE Architecture.** Because the MPA PEs are very different from SISD processors, performing execution-driven simulation is problematic.

- **Comparative Rarity.** Because MPAs are not in common use, developing and distributing an MPA to MPA execution-driven simulator has not been cost-effective.

These difficulties have not prevented substantial work from being done in MPA architecture research. In previous sections we showed that the following techniques have been in use: design from requirements, proof-of-concept by running representative sub-problems, and complete simulation. In the next section we will show the state of MPA benchmarks.

2.3.2 Benchmarks

As has already been mentioned, relative MPA performance on some particular tasks is dependent on the choice of algorithm. As a result, benchmarks have been almost exclusively task oriented. Because the importance of MPAs to image understanding, the benchmarks developed by the IU community have had great significance to MPA evaluation.

The Abington Cross benchmark [120] specification details a toy problem: Given a noisy regular geometric object, successively apply smoothing, thresholding, hole filling, and skeletonization operations. The result should be the medial axis transform of the original object. The simplicity of this benchmark has led to its wide application; however, its trivial nature made the results of questionable value. In the end, it has served mostly as an important stage in the development of the benchmark process itself.

The first DARPA benchmark [124] was also specifically vision oriented, but much more complex than the Abington Cross. It consists of ten tasks that can be categorized—by input data structure—into three types of problems that might be encountered at successive levels of the image understanding process: those dealing with 1) pixel arrays, 2) coordinate data, and 3) relational structures. This benchmark is also algorithm independent. One shortcoming that has been recognized in the DARPA I benchmark is that each task is self-contained and thus not representative of how a real image understanding system operates.

The second DARPA IU benchmark [148] addresses the problem of self-containment of tasks by specifying a high-level recognition problem as the overall goal of the computation, though specific sub-tasks are also specified. As a consequence, not only can results be obtained for both the overall problem and specific tasks, but it also reveals how long intermediate steps such as remapping of data take.

2.3.3 Particular Evaluation Systems

Now that we have discussed the general techniques in use in MPA architecture research, we review existing evaluation systems.

2.3.3.1 MAP Simulator

The Multi-Associative Processor (or MAP) was a multi-MPA. That is, it consisted of up to eight MPAs cooperating in parallel. Although the MAP simulator was built to tune a particular processor (the MAP), some of the ideas extend to the overall evaluation task [109]. The simulator was operable on two complementary levels: the array level and the system level. The array level simulator allows the user to vary the number of PEs and the amount of memory per PE. The system level simulator did not model instruction execution, but included details of system wide aspects of interaction among the constituent MPAs.

2.3.3.2 PAWS

PAWS is an interactive simulation environment for evaluating parallel processors [116]. It consists of four major components: the application tool, the architectural characterization tool, the performance assessment tool, and the interactive graphical display tool. The architectural characterization tool is of particular interest here. Within each class of machine (SIMD, MIMD, etc.), architectures are characterized based on number and flexibility of functional units, number of processors, memory characteristics, and interprocessor communication mechanism. Each category is partitioned hierarchically until the system is described at a fine enough level to be characterized by raw timing information. Either static values or dynamic values based on analytical models can be used. The performance assessment tool takes the output of the application tool (a data dependency tool) and maps it to the target architecture based on attributes. This research effort is still in progress.

2.3.3.3 SIMD Simulator Workbench

The SIMD Simulator Workbench [100, 99], developed at North Carolina State, is a complete simulator based on the BLITZEN [28]. The system has been used to examine four models of differing granularity, but with similar hardware requirements. They are MPAs with 1) 16K PEs with 1 bit ALUs, 2) 4K PEs with 4 bit ALUs, 3) 2K PEs with 8 bit ALUs, and 1K PEs with 16 bit ALUs. Also examined were the effects of adding specialized hardware support to accelerate multiplication and floating point instructions.

The applications examined are as follows: simulating a neural net, rotating an image, generating a Mandelbrot set, and image resampling. The results showed that for this set of applications, the smaller grained processors were favored. This is because many instructions are inherently single bit and because nearest neighbor moves are more costly for larger grained processors.

Some of the drawbacks of the SIMD Simulator Workbench are those inherent in complete simulation: the simulation is very time consuming and executions must be repeated for each change of parameters. Another limitation is the set of parameters that can be tried: for example, communication networks other than nearest neighbor connections are not supported; neither are variations in the PE memory hierarchy.

2.3.3.4 GT-RAW

The Georgia Tech Reconfigurable Architecture Workbench (GT-RAW) was developed to analyze reconfigurable architectures, so multiprocessor simulation is emphasized [95, 96]. However, among the possible configurations that GT-RAW is capable of evaluating are MPAs.

The GT-RAW combines the techniques of execution-driven simulation and ***virtual machine emulation***. The input programs are compiled to an architecture independent parallel code (AIC), which are then run on an AIC interpreter. As the instructions are executed, they are sent on to a trace analyzer where they are evaluated with respect to an architectural specification (ADC).

A more detailed description, paraphrased from [95], follows.

The AIC consists of segments of sequential stack machine code embedded with primitives that define the parallel structure of the program. Among them are

- Fork, Wait, and Exit which control thread creation and removal,
- Pealloc which defines SIMD segments of the program, and
- Send and Receive which process messages.

The interpreter is an event handler and interpreter of AIC instructions, i.e. an emulator for the AIC virtual machine. It generates a trace which is sent on to the trace analyzer. The trace analyzer adds architectural dependencies and creates and schedules events which are then fed back to the interpreter. The trace analyzer also models PEs, the communication links and the control units.

GT-RAW has been applied to parts of the second ARPA IU Benchmark [148]. Among the findings are that different parts of the benchmark are processed more effectively by processors of different types, demonstrating the advantages of processor reconfigurability.

One limitation of GT-RAW—which is largely a consequence of the fact that it supports distributed (MIMD) as well as centralized (SIMD) control—is that the process threads are executed separately by the virtual machine emulator. We will show later that a different approach to virtual machine emulation can yield an order of magnitude speed-up by taking advantage of the synchronous execution of MPA PEs. Another drawback, this one inherent in execution-driven simulation, is that (as in the SIMD simulator workbench) tasks must be rerun for every change in parameter. And finally, the detail of the evaluation is at a somewhat higher level than we would like. The memory hierarchy and the PE internals at the level of the instruction set are not considered. Also, the communication network simulator is based on a model that does not hold for all communication patterns.

2.3.4 Summary of Current Status

The state of the art in architectural evaluation is to use trace-driven or execution-driven techniques with real programs as the workload. The state of the art in MPA evaluation has lagged behind.

However, two of the systems just described—the SIMD simulator workbench and GT-RAW—represent a significant advance over previous work because of their use of real program executions in the evaluation of MPAs. They also have some limitations, however. One is the need to rerun simulations for every design change. Another is the range of features and parameters that it is possible to examine with the SIMD Simulator Workbench, and—because of its emphasis on broader questions in the domain of reconfigurable multiprocessors—the detail at which features are simulated by GT-RAW.

Studies that are more comprehensive than those just described—that is, studies which combine flexibility of parameter and feature selection, efficient and detailed simulation, reuse of computation, and large test suites—must overcome several problems as discussed in the introduction. These include simultaneously achieving accuracy and efficient simulation and creating a test suite containing real programs that does not skew performance toward any particular design.

PART II

ARCHITECTURE AND APPLICATION SPACES

Chapter 3

Architecture Space: Massively Parallel Arrays

Because we are primarily concerned with the architecture of massively parallel arrays (MPAs), this chapter presents, an overview of the MPA architecture space. The chapters that describe the evaluation of particular MPA components—such as the PE datapath, the PE memory hierarchy, and the interPE communication support—contain more details.

3.1 Overall Organization

The massively parallel arrays we consider in this thesis are characterized by the following minimum set of attributes:

- MPAs have SIMD control. This means that the system has two parts, a controller which broadcasts instructions and global data, and the array which is intended to perform the bulk of the computation.
- The number of PEs in the processor arrays typically ranges from several thousand to several hundred thousand PEs.
- PEs have their own registers and ALUs. The ALUs of existing systems range in size from a single bit to 32 bits.
- PEs have some amount of memory, either directly accessible, or from which they can be loaded, like, from a backing store.
- There is global-OR feedback from the array to the controller.
- PEs can communicate via an inter-connection network. For the purposes of this research it is assumed that the network has at least the topological sophistication of a mesh.

There are also several optional features which are present in some, but not all MPAs.

- **More complex PEs.** For example, PEs can have support for local indexing, or additional arithmetic hardware such as floating point support.
- **Additional feedback capability.** Some arrays have circuitry to return not just the global-OR of the values in a set of PEs, but also their number or the result of some other combining function.
- **Additional routing capability.** Common routing networks are broadcast meshes, multi-stage self-routing circuit switched networks, and packet switched networks.

See Figure 3.1.

In the next sections the major components of SIMD machines are discussed individually, after which some case studies will be presented.

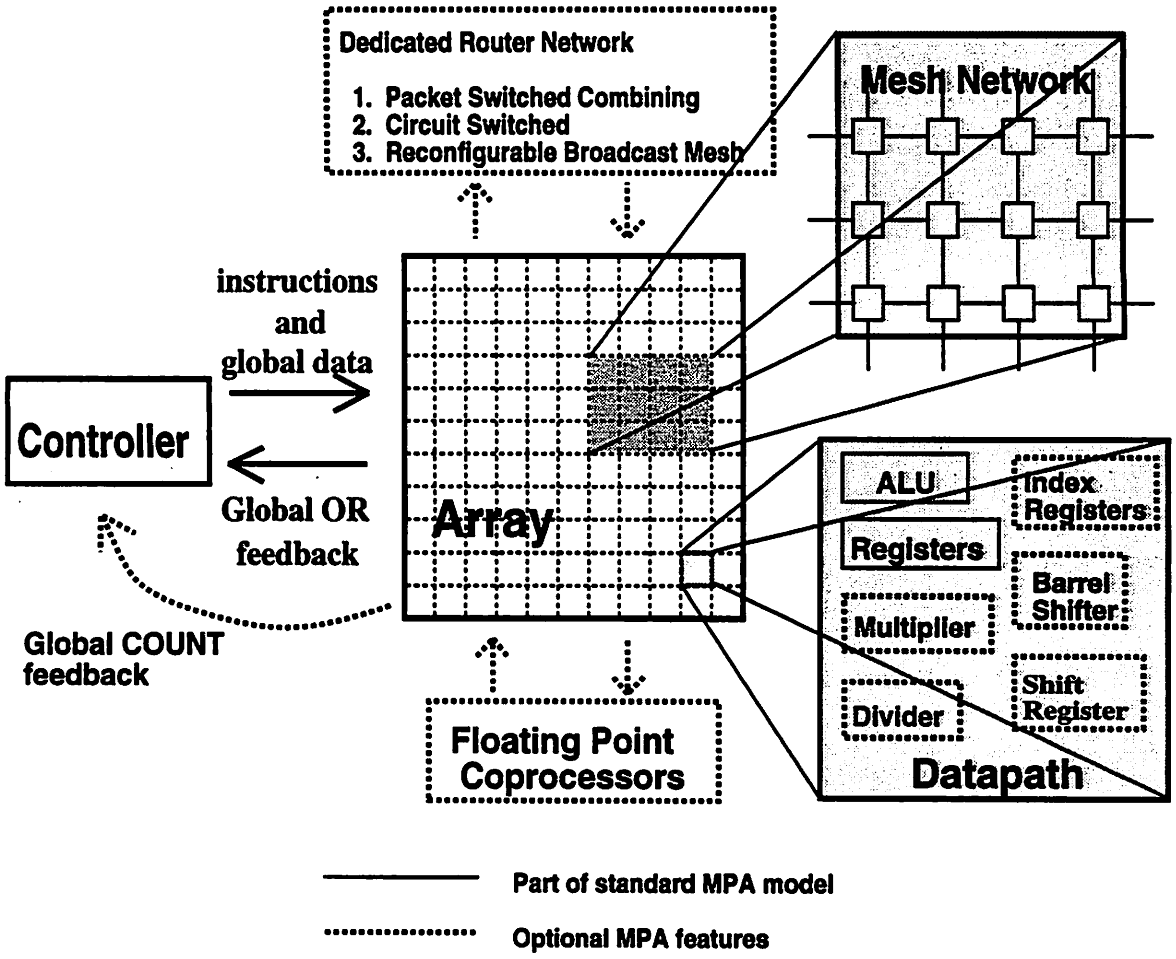


Figure 3.1. The MPA architectural design space.

3.2 Details of Particular Components

3.2.1 PE Internals

The PE designs in current MPAs are remarkably uniform, which is not surprising considering that the concept behind massively parallel arrays is to trade off complexity of the PEs to increase their number. The differences in the PE designs center around the datapath width, the number of functions supported, the number of address decodes per instruction, and the availability and level of floating point support. The latter can consist of a commercial floating point coprocessor, perhaps shared among some number of PEs as in the Thinking Machines CM-2 [137]. Alternatively, each PE can have its own support, somewhat short of a complete floating point coprocessor, such as the barrel shifter in the PEs of the MasPar MP1 and MP2 [103].

3.2.2 Memory Hierarchy Design

MPA memory hierarchy designs have also been very simple. Currently, there are two categories of storage:

- **On-Chip.** This storage (which we shall, for the sake of convention, call the *register set* or *register file*) can usually be accessed by the ALU in a single cycle. Most MPAs have at least some on-chip storage which typically ranges in size from 4 to 160 bytes.

- **Off-Chip.** Most MPAs also have a substantially larger off-chip memory which typically ranges from 2K to 128K bytes.

Other issues are whether indirect or indexed addressing support is provided. Some MPAs with this capability are the Blitzen and the MasPar MP-1 and MP-2 [28, 103]. Local indexing adds some local autonomy to each PE, although the response in current systems is substantially slower than direct memory access. Issues such as local addressing autonomy, and loading/unloading memory (the I/O bottleneck) are discussed further in [136].

3.2.3 Feedback and Associative Processing

In order for the controller to make data-dependent decisions in controlling the program flow (as is essential in real-time sensory processing), the array must contain hardware support to return rapid summaries. The most elementary summary is the return to the controller of the global OR of the value in a specified single-bit register across all PEs. This capability is sufficient to tell whether a data dependent computation has completed. A more complex summary is the return of the count of those register values. This additional information is used for efficient implementation of associative algorithms [58, 62, 144, 72] and for making data dependent algorithmic decisions [70].

3.2.4 InterPE Communication

All the processors considered have hardware support for inter-PE communication among nearest-neighbors in a two dimensional mesh configuration. We also consider some processors that have more sophisticated communication networks; these fall into the following categories: packet switched, circuit switched, broadcast or wormhole.

- **Packet Switched Networks** The controller directs inter-PE communication by having each PE construct a packet containing the information to be sent and the ID of the receiving PE. These packets are then transmitted over the network where they are buffered at each intermediate node between the source and the destination. Therefore, only the next node need be open for the packet to make progress; if the packet is blocked it remains where it is until the next node is ready to accept it. For example, the CM-2 has such a communication network [137]. It is also worth noting that any processor with support for nearest neighbor communication can simulate such a network [70].

- **Circuit Switched Networks** Circuit switched communication differs from packet switched in that the entire path between source and destination must be clear before the packet can be sent. The information is then sent point-to-point, directly from source to destination. The paths can either be created by a global controller with knowledge of the communication pattern, or can be created 'on-the-fly' by packets as they traverse through the network. Such *self-routing* circuit switched networks are the only viable alternative for massively parallel processors to route arbitrary permutations: the cost of calculating the routing paths off-line would be prohibitive. Self-routing networks have the problem that PEs do not know before they transmit a packet whether a path is open all the way to the destination. Since a blocked packet will be lost in such a network, a mechanism must exist for the destination to acknowledge to the source that the packet has indeed arrived. The MasPar MP1 and MP2 have self-routing circuit switched networks [103].

- **Broadcast Networks** Broadcast networks are characterized by the fact that PEs are all connected to a bus or buses that connect a row, a column, or the entire array of PEs. PEs communicate by writing data onto a bus and letting it propagate, following the underlying topology of the network. PEs connected to the network then selectively read the information. In some systems, if multiple PEs broadcast on a circuit simultaneously, the wire-OR is transmitted. Commonly, PEs also have control of switches through which they can disconnect a nearby portion of the network. Using this mechanism, some implementations enable the array to be partitioned into any number of arbitrarily shaped contiguous broadcast buses. At least three varieties of broadcast networks have been, or are being, built. They are all based on the two-dimensional mesh topology.

  - **Broadcast Buses** [114, 113]. PEs can broadcast/receive data to/from their own rows/columns.

  - **Reconfigurable Buses** [93, 94]. These are the same as broadcast buses, with the added capability that PEs control switches to open circuit the bus in either direction on either bus, preventing the signal from propagating further.
  - **Coterie Network** [147]. The coterie network is also known as the reconfigurable mesh. It is similar to reconfigurable buses, except that signals are not restricted to propagating down only the rows or the columns, i.e. the rows and the columns can be shorted together. In this way arbitrary contiguous aggregates of PEs can be electrically connected.

- **Wormhole Routers** Virtual cut-through was developed by Kermani and Kleinrock [84] as an alternative to routing via circuit switching or store-and-forward packet switching. They state:

  > When a message arrives at an intermediate node and its selected outgoing channel is free, then the message is sent out to the adjacent node towards its destination before it is received completely at the node; only if the message is blocked due to a busy output channel is a message buffered in an intermediate node.

  The great advantage of this method is that the overhead of buffering the message at every node is eliminated. Dally and Seitz [50] modify cut-through routing (and rename it wormhole routing): packets are divided into a series of 'flits.' When the head of a packet is blocked, the rest of the packet is not queued in that intermediate node, rather the trailing 'flits' remain where they are, occupying their current channel until they are allowed to continue. A hardware wormhole router chip has also been implemented by INMOS for use as a building block in creating transputer communication networks [126].

There are many further levels of distinction among networks, the most important being synchronous versus asynchronous and how the nodes are connected or the network topology. Discussions of these and other aspects of communication networks can be found in many different surveys, such as [59, 37, 7].

3.3 Case Studies

Massively parallel arrays have been very popular in the research literature, with the initial ideas formulated by Unger [142] and Holland [75]. This section contains very brief presentations of some of the more recent and more influential MPAs. Machines that are now mostly of historical interest can be reviewed in Weems's dissertation [144]. More details of many of these machines can be found when the particular architectural components are discussed in Chapters 8, 9, and 10.

- **Abacus [30].** Abacus is a fine-grained MPA whose PEs have one bit wide ALUs. The memory consists of 32 bits on-chip and 16K bits off-chip. The routing networks are a nearest neighbor mesh and a reconfigurable broadcast mesh. No feedback mechanism is specified, but a global OR is trivial given a broadcast network.

- **Blitzen** [28]. The designers of the Blitzen used the MPP as a starting point (see below), so the one bit ALUs are similar. The Blitzen has 1024 bits of on-chip memory per PE, larger (though unspecified) off-chip memory and indirect addressing support to give PEs local addressing autonomy. The interPE communication is via X-connections, again similar to the MPP, except that they support eight-way moves. There is feedback support for global OR.

- **Content Addressable Array Parallel Processor** or **CAAPP** [147]. The CAAPP PEs have one bit ALUs, but 8 bit data paths for intraPE data transfers. There are 320 bits on-chip and 32K bits off-chip memory. InterPE communication is via a nearest neighbor mesh and a reconfigurable broadcast mesh. There is feedback support for both global OR and global Count.

- **CLIP4** [63]. The CLIP4 PEs have one bit wide ALUs and 32 bits of local RAM. There is also staging memory of unspecified, but far larger, size. The CLIP4 has support to run in bit-parallel mode. That is, rows and columns of single bit PEs can emulate 32 bit PEs. Communication is via a 6 or 8 connected nearest-neighbor mesh and carry propagation (effectively a broadcast mechanism) along rows and columns. There is feedback support for global Count.

- **Connection Machine** or **CM-2** [73, 137, 140]. The CM-2 PEs have one bit wide ALUs, share a number of floating point coprocessors, and have access to 64K bits off-chip memory. Routing is via a dedicated packet switched communication network that has a truncated hypercube topology. Local (NEWS) and power-of-2-away communication have significantly faster execution times than arbitrary communication patterns. Also, the router chip is used to accelerate some operations requiring indirect addressing. There is feedback support for global OR.

- **Distributed Array of Processors** or **DAP** [79, 113]. The DAP PEs have one bit wide ALUs; there is also hardware support to run in bit-parallel mode. That is, rows and columns of single bit PEs can emulate 32 bit PEs. PEs have no on-chip memory, but have 32K bits high-speed RAM off-chip. Communication is via a nearest-neighbor mesh plus vertical and horizontal broadcast buses. There is feedback support for global OR.

- **Data Transport Computer** or **DTC** [81]. The DTC PEs have a one bit wide ALUs, 2K bits of on-chip and from 8K to 32K bytes off-chip memory. InterPE communication is via a 3 dimensional nearest-neighbor mesh. There is feedback support for global OR.

- **Geometric Arithmetic Parallel Processor** or **GAPP** [44]. The GAPP PEs have one bit wide ALUs, 128 bits on-chip, and an unknown amount of off-chip memory. InterPE communication is via a nearest-neighbor mesh. Feedback support is not known.

- **GEC Rectangular Image and Data computer** or **GRID** [114]. The GRID PEs have one bit wide ALUs and 64 bits dual-ported on-chip memory. InterPE communication is via an 8 connected nearest-neighbor mesh and row and column broadcast buses. Feedback is supplied to the controller by registers which maintain the OR of the row and column buses.

- **MasPar MP-1** and **MP-2** [65, 103, 25, 108]. The MasPar MP-1 PEs have 4 bit wide ALUs while the MasPar MP-2 has 32 bit wide ALUs. The MasPar PEs also have support for floating point operations in the form of a barrel shifter and floating point registers. PEs have 1536 bits of on-chip and 16384 bytes of off-chip memory. InterPE communication is via X-Connections (effectively an 8 connected nearest-neighbor mesh) and a circuit switched permutation network. There is feedback support for global OR.

- **Massively Parallel Processor** or **MPP** [18, 19]. The MPP PEs have one bit wide ALUs and shift registers which are especially useful for multiplication instructions. PEs also have 1024 bits local, though off-chip, memory. InterPE communication is via a nearest neighbor mesh. There is feedback support for global OR.

- **Polymorphic Torus** [93, 94]. The Polymorphic Torus PEs have one bit wide ALUs and 256 bits of on chip memory Communication is via a mesh with reconfigurable broadcast buses. No feedback support is specified, but a global OR is trivial given the routing support.

- **Sliding Memory Plane Array Processor** or **SliM** [134]. The SliM PEs have 8 bit wide ALUs. The memory is not specified. InterPE communication is via a nearest neighbor mesh with some by-pass circuitry to accelerate window-based operations. There is feedback support for global OR.

3.4 Conclusions

Massively parallel arrays have certain characteristics in common: they are divided into controller and array, the control is SIMD, the arrays are composed of large numbers of simple PEs, there is an interPE communication mechanism, and there is some feedback capability between array and controller. Architectures can differ in the size of the array, the complexity of the PEs, the amount of memory on- and off-chip, the type of communication network, the topology of the communication network, the internal and external data path widths, the presence/absence of local memory indexing, and the type of feedback. While other variations, such as local control, are possible, we chose to limit the focus of this research to the aforementioned architectural design issues for purely practical reasons. These additional variations can form the basis of a considerable amount of future research.

Chapter 4

Application Space: Spatially Mapped Applications

4.1 How to Choose a Test Suite

The fundamental criterion for selecting the test suite programs is that they should be representative of the expected workload to be run on the target machines. In the absence of precise information about the workload, certain criteria should be used to guarantee the usefulness of the results. These are as follows:

- The programs should be derived from the application domains where the target machine will be used. If the designs being evaluated are still experimental, then there should be some indication (e.g. from requirements analysis) why it is likely that the architectural class in question is a cost-effective alternative in that domain.

- The programs should be in use (rather than toy programs). Thus the results the evaluations will have some validity as components of the workload, even if the precise workload distribution is not known.

- The programs should be sufficiently complex to exercise the target machine appropriately.

- The types of computations represented by the test suite should, as much as possible, span the space of types of computations likely to be encountered in practice. This is essential to ensure that all likely computations will be supported adequately.

In the next section, applications for which massively parallel arrays have proved to be a good alternative are enumerated and types of computations encountered within these tasks are discussed. The test suite is then presented.

4.2 MPA Application Domains: Current and Future Workload

It is an intractable problem to perform a detailed study of all computationally intensive applications and to thereby determine onto which architectural class any particular application would be most cost-effectively mapped. The fact that there are MPA vendors competing with vendors of other classes of architectures for use on the same types of applications should be evidence that enough information is not yet available to fully answer this question. Further, the recent proliferation of work in heterogeneous processing is recognition that no single class of architecture is likely to be most cost-effective for solving all problems, even those that are computationally intensive.

The goal of this section is to propose a likely workload for the target machines to be specified in this study. This is done in two parts. In the first, cases are presented where users—who have a choice of processor for that particular application—are using MPAs. These applications are a true indication of typical MPA workload. The second part discusses one of the most promising future application domains where MPAs will be applied: computer vision. It is shown that key components of computer vision processing are very well suited to execution on MPAs.

4.2.1 Where MPAs Are Used

MPAs have been found to be cost-effective processors in many areas of computation. Some of the applications for which particular machines have been used and the types of computations that dominate these applications are now described.

- **Image Analysis.** One of the earliest MPAs, the Illiac III, was designed to process bubble chamber negatives [107]. The input consisted of binary images of ion tracks. The output was a list of nodes (defined as end-points and branches) that characterize each track. The preliminary tasks in processing the tracks are computing edges, line thinning, gap filling, and line smoothing. To obtain the node information about each track, endpoints and branchpoints of each track are identified and correlated.

  The Goodyear MPP has long been used by the Goddard Space Flight Center for satellite and shuttle image analysis [132]. Among the specific tasks is the determination of elevations from stereo images. The primary operations used in this process are image warping and normalized correlation with various window sizes.

- **Signal Processing.** A STARAN and its successor, the ASPRO, have been used for early warning radar surveillance and command and control processing [19]. One major function is to process and correlate targets to form track data to be output to operator displays.

- **Scientific Modeling.** A DAP is being used at the University of Edinburgh for the simulation of physical systems [150]. Among those phenomena modeled are the spread of forest fires and disease, and thermal phase transitions. These processes all make use of relaxation-type algorithms.

- **Finite Element Simulation.** A MasPar MP-1 is being used by NASA Goddard for finite element simulation of solid mechanics and fluid flow [60]. Systems are simulated by discretizing a continuous model of the physical world into a collection of finite elements and processing each element based on its neighboring elements. Since the relationships between elements is unstructured, irregular communication must be supported.

- Cartography
- Chip Routing
- Climate Modeling
- Data Retrieval
- DNA Sequencing
- Fluid Flow
- Hydraulics Modeling
- Image Analysis for Astronomy and Earth Science
- Image Processing
- Lattice Gauge Theory
- Logic Simulation
- Molecular Modeling
- Neural Networks
- Signal Processing
- Speech Recognition

Table 4.1. Partial list of applications being run by end-users on the Cambridge Systems DAP.

- **Rendering and Light Simulation.** A MasPar MP-1 is being used by NASA/Goddard to create perspective images of data sets consisting of hundreds of thousands of points [60]. This process is computed in two steps. In the first step, the projection of each data point is calculated. The data points have been distributed arbitrarily among the processors and this computation is performed fully in parallel at each PE using global data. The second step requires the brightness and range information to be sent to the correct screen positions.

  A Thinking Machines CM2 has been used for the computer graphics problem of simulating light propagation and its interaction with matter [87]. The basic technique is a three dimensional relaxation algorithm.

These applications are among the many described in the literature. See especially the Proceedings of the Symposia on the Frontiers of Massively Parallel Computation for many others. More recently, however, MPAs have been in use more and more in industrial settings where the particulars of the application are not publicized. Some of these applications are described by the MPA manufacturers. For example Active Memory Technologies (now Cambridge Systems) describes the use of the DAP in the applications found in Table 4.1 on Page 40 [5].

MasPar goes one step further, describing areas where commercial application software has been developed for the MP1 [105]. These can be found in Table 4.2 on Page 41.

4.2.2 Where MPAs Will Be Used: Computer Vision

One of the most promising areas to which MPAs have and will continue to be applied is computer vision. Innumerable studies have been published that have shown the advantages, if not the necessity, of using MPAs for certain aspects of vision processing. However, the problem

- Crash Simulation
- Finite Element Analysis
- Fluid Dynamics
- Statistical Analysis
- Text Search

Table 4.2. Partial list of applications that have been developed by third-party vendors to run on the MasPar MP1 and MP2.

in characterizing exactly what these particular tasks are is problematic: thirty years of research has shown that vision is a very complex *domain* composed of many distinct problems (image understanding, navigation, inspection, identification, etc.). Each of these problems requires performing dozens if not hundreds of distinct sub-tasks.

One difficulty is that since most of these computer vision problems have not been completely solved, no one can be certain what a true vision workload will be. There is, however, a great deal of consensus as to the essential components of vision processing, as well as certain tasks, algorithms, paradigms etc. that are almost certain to be a major part of a vision workload. And because the promise of vision processing is so immense, and the amount of computation so huge, a great deal of thought has been given about how to map these sub-tasks to various massively parallel processors.

In the rest of this subsection we will characterize the MPA workload that is likely to arise in computer vision applications.

4.2.2.1 Computer Vision Paradigm

The following description is paraphrased from Rosenfeld [123]. The goal of image analysis (one of several tasks that comprise computer vision) is the construction of scene descriptions on the basis of information extracted from an image or image sequences. The basic steps of this process are to extract information from the image(s), build it into relational structures, and match those structures against similar representations (models) stored in memory.

Even from this broad description of the recognition problem, certain aspects of the computational requirements can be inferred: one essential task is obviously model matching, another the extraction of symbolic information from the input image. Slightly less obvious is that it is often convenient to interpose intermediate levels of symbolic processing between these two tasks (such as those facilitated by the Intermediate Symbolic Representation, or ISR [36]). Also, it is well known that vision is an underconstrained problem, that scene information is necessarily lost in the image formation process. Computer vision systems attempt to deal with having insufficient data by using multiple passes through the data, using feedback, applying stored knowledge, etc.

4.2.2.2 *Computational Requirements of Computer Vision*

The generic vision task description in the previous section is sufficient to show that the computational requirements are immense: processing even elementary operations on images at frame rate requires the execution of hundreds of millions of instructions per second. Estimates have been made that a rate of execution several orders of magnitude higher than that will be needed to perform real-time image understanding [145].

These figures indicate that the appropriate question is not whether massively parallel processors must be used for computer vision processing; rather what they should look like. And given the price/performance advantage of MPAs over other classes of architectures, the question becomes whether any parts of the vision problem map efficiently onto MPAs.

4.2.2.3 *Computer Vision Tasks Suitable for MPAs*

The parts of the computer vision domain that have most often been assigned to MPAs are referred to as low-level vision, which is generally meant to denote those tasks where the input is an image or its transformations. The justification for this assignment is derived from the obvious mapping of image pixels onto PE arrays, from many years of experience in developing applications for MPAs, and from the results of numerous benchmarks. The last point is discussed in detail at the end of this section. First we describe the tasks themselves.

It is beyond the scope of this proposal to present an extensive survey of low-level vision as probably hundreds of papers are published in this area every year. It is critical to our study, however, to try and characterize the processes involved.

Low-level vision is necessarily not completely specifiable because its bounds are artificial; it has been defined in many ways:

- Marr describes extraction of a primal sketch as making "explicit important information about the two-dimensional image, primarily the intensity changes there and their geometrical distribution and organization [101]."

- According to Brady, low level processing was historically "more art than science, and largely consisted of methods for the extraction of 'important' intensity changes in an image." However, he also devotes a section to "modules that operate on the image." [33]

- Ballard and Brown refer to "early processing" as the set of computations by which the degeneracies of the imaging process are undone [15].

- Zucker is even more vague, referring to "early (low-level) vision" as "those problems for which the solution is driven by general-purpose assumptions and special-purpose hardware [152]."

- Weems defines low-level vision as sensory processing [145]. In particular, low-level vision tasks are those whose input representations are images either of input sensory data, or other spatially distributed (iconic) image events such as edges, regions, etc.

We choose to use the last definition and refer to low-level vision as being the processing required to map images onto images, and to reduce images to symbolic descriptions. In some nomenclatures, this latter process is referred to as intermediate-, or even high-level vision. We reserve those terms for operations on symbolic databases and knowledge based vision respectively.

Low-level vision should not be viewed in a vacuum, however, since it is of little use if not integrated into a larger system. Although it consists mostly of semantically independent operations on the original input image and other image-like data derived from that image [17], we must allow for the fact that one must be able to apply knowledge to aid in feature extraction. Details and compendia of specific low-level vision operations can be found in a large number of surveys [33, 101, 15, 123, 152, 145], and as well as in some benchmark specifications [124, 148].

4.2.2.4 Low-level Vision Processing Applications

Some of the functions considered to be part of low-level vision (at least some of the time) are as follows: edge detection, line extraction, region segmentation, extracting surface parameters, motion detection, stereo matching, and extracting texture measures.

- **Edge detection operators.** The principle behind edge detection is that discontinuities in the image tend to be caused by depth, orientation, illumination, reflectivity, and/or color in the scene. The way to detect discontinuities in an image is to take a derivative. Many standard operators are used; they are typically based on the application of a mask. It has also been found that better results can sometimes be obtained by using the second derivative, either alone or together with the first derivative [40], or by combining derivatives with filters [102].

- **Grouping.** The principle behind grouping proximate pixels with some value in common is that these groups, segments, regions, etc. tend to have some meaning in the scene. Grouping tends to be divided into those algorithms that group input spectral values into regions [110, 22], and those that group discontinuities into lines [38, 29], or, as in Boldt's work, the grouping of higher order structures such as edges into lines [29].

- **Motion.** There are many popular methods for determining motion, either of the observer or of the objects in the scene, through the analysis of sequential image frames. These include using feature correspondence, optical flow, and direct matching of image intensities [78]. One particular method combines the establishment of correspondences through a hierarchical correlation algorithm with the application of a gradient-based smoothness constraint to filter possibly erroneous depths [9].

4.2.2.5 Low-level Vision Benchmarks on MPAs

As stated earlier, results of numerous benchmarks have demonstrated the advantages of using MPAs to process low-level vision tasks.

The Abington Cross benchmark specification [120] details the following problem: Given a noisy regular geometric object, successively apply smoothing, thresholding, hole filling, and skeletonization operations. The machines with the best performance were MPAs such as the CM-1, GAPP, and DAP. When cost factors were included the AIS5000 one dimensional array also became an attractive choice. The one dimensional array performed well because the entire benchmark could be executed using window operators. The simplicity of the Abington Cross benchmark made it widely applied; however, its simple nature made the results of relatively little value.

The first DARPA benchmark [124] consisted of ten tasks that could be categorized by input data structure into three types of problems that might be encountered at successive levels of the image understanding process: those dealing with 1) pixel arrays, 2) coordinate data, and 3) relational structures. Generally the MPAs such as the Connection Machine performed the best, although the Warp, a systolic processor [11], also had impressive performance.

One shortcoming of the first DARPA benchmark was that each task is self-contained and thus not representative of how a real image understanding system would operate. The second DARPA IU benchmark [148] addressed the problem of self-containment of tasks by specifying a high-level recognition problem as the overall goal of the computation, though specific sub-tasks were also specified. As a consequence, not only were results obtained for both the overall problem and specific tasks, but it was also revealed how long intermediate steps such as remapping of data would take. Again, the MPAs had superior performance on the low-level vision parts of the task.

What the two DARPA benchmarks showed was that MPAs are not only an excellent choice for the window-based image processing applications dominant in the Abington Cross, but also for spatially mapped tasks with a variety of communication requirements.

4.3 Characterizing the MPA Workload

The applications described above can be characterized qualitatively by the following:

- They are computationally intensive.
- They involve processing large amounts of data.
- Effective algorithms can be found that require few threads of control.
- The data are often derived from real world processes.
- It is usually advantageous to map the data to an MPA in a such a way so that the spatial relations inherent in the problem are preserved among the data in the array.
- There is often spatial locality in the effect of data on its neighbors.

Most of the types of computation performed by the MPAs fall into one of the following categories:

- **Array Operations.** These are arithmetic operations performed on the array elements in parallel. They are an integral part of all computations that map efficiently to MPAs. Array operations typically operate on either integer or floating point data.
- **Neighbor Operations.** Many computations proceed iteratively with array and nearest-neighbor data exchange operations alternating on each iteration. Neighbor operations are particularly important in the applications that require filtering or that use relaxation algorithms.
- **Operations On Connected Components.** These operations are used to characterize data mapped to contiguous sets of PEs. This is necessary whenever multiple objects or parts of object hypotheses in an image are processed in parallel. Also, connected components in the data often have immediate semantic content, e.g. bubble chamber tracks that must be characterized. Since connected components are often referred to as regions, we also refer to these operations as region-based.
- **Operations Among Connected Components.** These operations are an essential part of grouping algorithms where components that share some common characteristic are merged.
- **Feedback Operations.** Extraction of data from the array by the controller is an essential component in all data-dependent algorithms. Feedback is also critical in associative processing that takes place in text processing, data retrieval, and image analysis applications.
- **Permutation Routing.** In permutation routing, data in each PE is transferred to another distinct PE. The packet destinations can often be determined locally within each PE by a data dependent computation. This operation is used in applications, e.g. those in graphics, that involve data mapping and in various nonimage transformations such as the Fourier.

From the preceding discussion it is apparent that the computations for which MPAs are suited are not just those that transform one image into another, but also those where one or more parts of the computation operate on other forms of spatially mapped data. This means, for example, executing communication operations that are not just regular, but also non-uniform; not just within windows, but over arbitrary distances in the array; and not just one-to-one, but also broadcast, reduction, and parallel-prefix.

4.4 Representing the Workload: the Test Suite

As mentioned previously, our criteria for selecting the test suite are that the tasks should: be representative of the application domain, and preferably in use; be significant applications, rather than simple operators; exercise the target architecture; and span the space of types of computations encountered in the domain.

From the discussion in the previous section, it follows that a set of programs derived from the domain of *spatially mapped applications* would be appropriate. By spatially mapped applications we mean those applications derived from real-world processes where the data are mapped to the processor array in such a way so as to preserve the spatial relations inherent in the application. This domain includes tasks from the applications mentioned above, including image processing and analysis, pattern recognition, low-level computer vision, finite element modeling, simulation of physical processes, engineering applications such as routing for VLSI and printed circuit boards, and many others.

Although the test suite programs listed below primarily involve vision tasks, many of the characteristics are common across the entire domain.

In the following descriptions, it is assumed that images are mapped to PE arrays one pixel to one PE. In the probable event that the PE array is not sufficiently large to provide this mapping, pixels are still mapped one-to-one conceptually, but to *virtual* rather than physical PEs. The physical PEs emulate the appropriate number of virtual PEs.

4.4.1 The ARPA Benchmark

The second ARPA image understanding benchmark was developed to provide a tool with which to evaluate parallel processors [148]. The key idea is that the benchmark specification should consist of an integrated series of tasks that collectively require the same range of data types and computations found in the execution of a true vision task. The low- and intermediate-level components of the benchmark were found to map extremely well to MPAs, and so are used in this benchmark suite. The constituent tasks are as follows.

1. Label connected components.
2. Compute K-curvature.
3. Extract corners.
4. Select components with three or more corners.
5. Determine the convex hulls of corners for each component.
6. Compute angles between successive corners on convex hulls.
7. Select corners with K-curvature and computed angles indicating a right angle.
8. Label components with three contiguous right angles as candidate rectangles.
9. Compute size, orientation, position, and intensity of each candidate rectangle.

In all tasks, besides labeling the connected components and determining convex hulls, the application to MPAs is straight-forward. The tasks are computed using array, nearest-neighbor communication, and region-based operations. The other two tasks are sensitive to the routing network support and are described in detail later. The input image used as input for the benchmark throughout this work is shown in Figure 4.1.

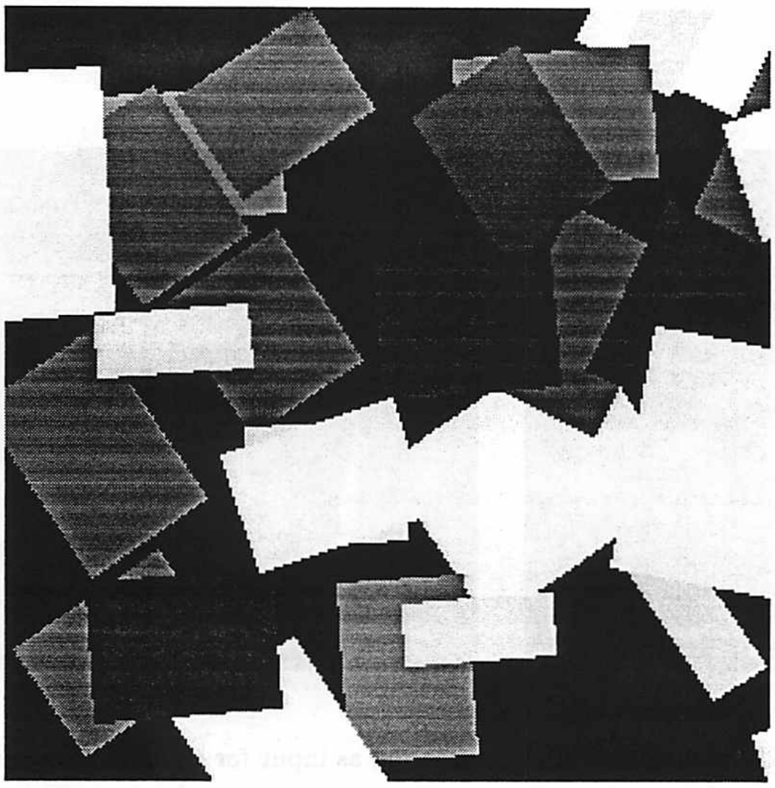


Figure 4.1. Synthetic 256x256 8 bit gray scale image used as input for the second ARPA IU Benchmark.

4.4.2 Correlation-based Correspondence

The correspondence problem in vision is as follows: given two images I_1 and I_2 of the same scene (usually a stereo pair or two images obtained from slightly different viewpoints due to camera motion) find the points in I_1 and I_2 that correspond to the same points in the scene. One way the correspondence problem is addressed, for a given point (x_1, y_1) in I_1, is to try out proximate points in I_2 and see which match the best. For example, if the maximum disparity of corresponding points is known to be 5, a window with size 10×10 is specified. Then the neighborhoods around those 121 points in I_2 (i.e. $\{(x_2-5, y_2-5), (x_2-4, y_2-5), \ldots, (x_2+5, y_2+5)\}$) are all correlated with the neighborhood around (x_1, y_1) in I_1. Typical correlation neighborhoods are 3×3 or 5×5. The point in I_2 which correlates best with (x, y) is saved. This process is repeated for every point in I_1. The correlation computation itself consists of a sum of point-for-point differences for all points in the correlation neighborhood. See [15, 57] for details.

Parallelization methods for correlation algorithms are well known. The basic idea is for each correlation window to be moved around the correspondence window in a spiral fashion. The operations that dominate are mesh communication and the addition and subtraction array operations. The images used as input for the correlation procedure throughout this work are shown in Figure 4.2

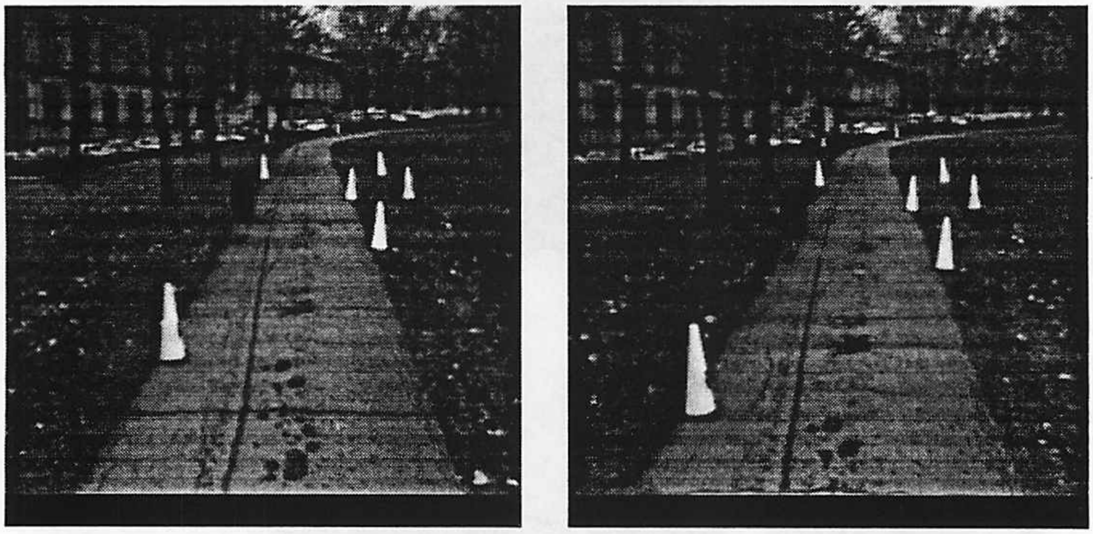


Figure 4.2. Two 256x256 8 bit gray scale images used as input for the correspondence matcher.

4.4.3 Weymouth-Overton Edge-Preserving Filter

Essential to many vision tasks is preprocessing to smooth images. The objective is to remove noise while enhancing, rather than blurring, spectral discontinuities. One algorithm that provides this function was developed by Overton and Weymouth [111]. The key idea is to use an empirically derived non-linear edge model. Areas of the image that are found likely to be edges are enhanced to make them more like the model, all other areas are smoothed to make them look less like the model.

The basic method is for each pixel to obtain the weighted average of its neighbors over multiple iterations. As the iterations progress, the weights are varied so as to favor the most similar neighbors.

The parallel version of the Weymouth-Overton filter resembles the correspondence function in that it also uses mesh communication. It differs, however, in the complexity of the operations at each point. Each PE must perform several floating point operations between communication operations. The image used as input for Weymouth-Overton filter is shown in Figure 4.3.

4.4.4 Region Merging Segmentation

A modified version of the region merging phase of the Nagin-Kohler segmentation algorithm [22] is another test suite component. Since a description of the parallel version of this algorithm has not yet been published we present it here in more detail than the other test suite programs.

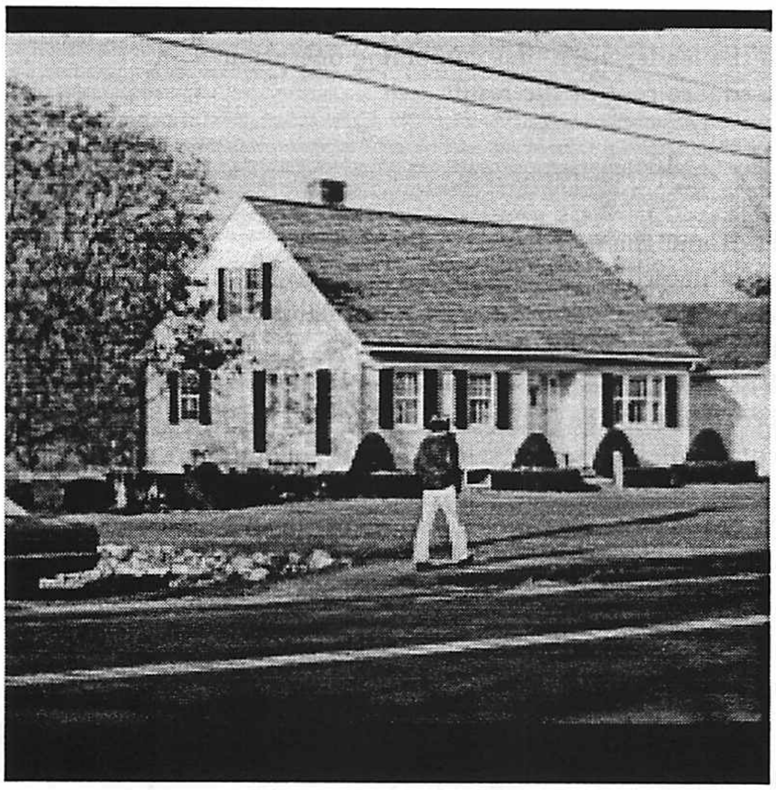


Figure 4.3. The 256x256 8 bit gray scale image used as input by the two line finders and the curve-fitting filter.

The region merging phase begins by characterizing each region with the following attributes: the size, the means and standard deviations of various spectral quantities, and by the lengths of its common borders with adjacent regions. Based on these values, merge scores are calculated for each pair of adjacent regions. Region pairs are merged if their merge score is both a local minimum and lies below a global threshold. After the merge is completed, the newly created regions are again characterized and new merge scores calculated for region pairs. The process is repeated until no merge scores surpass the global threshold.

The bulk of the computation is performed in the following procedures.

GetRegionSize
1. Each region selects a unique master or leader PE.
2. Every PE in the region sends a one to the leader via a plus combining operation.

GetAverageRegionIntensity
1. GetRegionSize is run.
2. Each PE sends its intensity to the region leader via a plus combining operation.
3. The region leader divides the sums of the intensities by the region sizes.

```
GetRegionIntensityStandardDeviation
1. Each PE sends the square of the difference between its intensity and the
   average intensity to the leader via a plus combining operation.
2. The leader takes the square-root of the result.
```

```
GetCommonBorderLengths
1. Border PEs fetch the IDs of the neighbor regions.
2. While there are any unselected border PEs in any region:
3.       Each region selects the set of border PEs who's neighboring ID is the
         maximum of those still unselected.
4.       The selected PEs are counted using a plus combining operation.
5.       The count is broadcast to the region.
6.       The count is communicated to the selected PEs' neighbors in the region
         who's ID was selected.
7.       The PEs which have either just been counted, or who's neighbors have just
         been counted, are removed from further consideration.
```

The above procedures are all straight-forward, except GetCommonBorderLengths whose performance depends on the adjacency graph. It can be shown, however, that since the region adjacency graph is planar, the minimum number of regions required to force this procedure to go through i iterations is

$$2 * \sum_{j=1}^{i-1} \frac{(i-1)!}{(i-j)!}$$

This means that convergence is achieved after 4 or 5 iterations for likely image sizes.

Beyond the obvious parallelization achieved by mapping the input image to the processor array and of using region-parallel operations, the major difference between the sequential and the parallel versions of this algorithm is that the number of iterations has been drastically reduced in the parallel version. By modifying the merge policy slightly, we have found that the number of iterations can be reduced from hundreds, in the original version, to less than 20 with little loss in segmentation quality.

The image used as input for the region merging segmentation procedure throughout this work is shown in Figure 4.4.

4.4.5 The Daumueller Line Finder

Most line finders depend on grouping edge elements that have been computed by taking the gradient magnitude of an image. The Daumueller line finder, however, uses the approach developed by Burns of grouping edge elements based on gradient *orientation* [52]. The key steps (described in detail in [38]) of the original Burns algorithm are as follows.

1. Proximate pixels of similar gradient orientation are grouped into line-support regions.

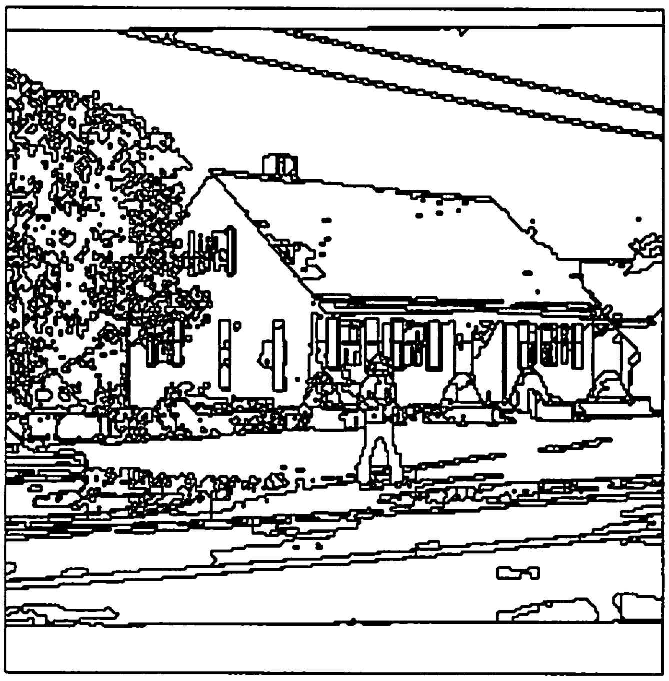


Figure 4.4. Edge image corresponding to the 256x256 32 bit region label image of a partial segmentation of the gray scale image shown in Figure 4.3. Used as input for the region merging segmentation procedure.

2. Within a line-support region, line attributes such as direction, length, and contrast are computed. The method is to fit a plane to the gradient magnitude depth map of the region. This line orientation is obtained by intersecting the fitted plane with a horizontal plane representing the average intensity of the region weighted by the local gradient magnitude.

The second step proves to be much more computationally intensive than the first. A more efficient alternative has been developed by Daumueller. We describe the algorithm for one line support region; the same procedure is actually executed in all regions simultaneously.

The region is divided into three subregions as follows. The extreme points along the axis perpendicular to the gradient orientation are selected and define a line segment. The line segment is then trisected with two perpendicular lines. These perpendicular line segments (that are parallel to the gradient orientation) and the original region boundary form the boundaries for the three sub-regions. Within each subregion, the three points with the greatest gradient magnitude are selected. A least squares fit of these 9 points is computed. The two points where this line intersect the region boundary specify the line segment denoted by the line support region.

The image used as input for Daumueller line finder is shown in Figure 4.3.

4.4.6 Fast Line Finder

An alternate version of the Burns algorithm is the so-called Fast Line Finder [83]. The version we use was parallelized and tuned by J. Burrill.

First the line support regions are formed as before. A line is then fit to the support region by obtaining its principle axis. The principle axis is determined from the eigenvalues computed from a scatter matrix (see [83] for the constraints). Again, the two points where this line intersect the region boundary specify the line segment denoted by the line support region.

The image used as input for the Fast Line Finder is shown in Figure 4.3.

4.4.7 The Dutta Depth-From-Motion Procedure

Dutta's depth from motion procedure takes as input the correlation and the correlation reliability of an input image pair. The following description of the procedure is derived from [57].

The process requires four stages.

1. **Selection of the best image displacements.** The image is partitioned into pre-determined, equal area, sub-regions. In each sub-region, the pixel which has the most reliable displacement is selected. The focus of expansion (FOE) and motion parameters are determined for those vectors.

2. **Determination of approximate translation.** A line through each of the selected vectors is computed. All the intersections are plotted using the Hough transform. The area in the Hough space which contains the maximum number of intersections indicates the proximate location of the FOE.

3. **Determination of exact translation and rotation.** Once the FOE placement has been bounded, the exact translation and rotation parameters can be computed by using the following optimization method.

   For each possible FOE position (in the bounding window), the normalized absolute deviation in directional depths k is computed for the rotations. The rotations corresponding to the minimum k are optimal. The minimum k is determined from among the FOE hypotheses. The translation and rotation corresponding to the minimum k are the exact motion parameters.

4. **Depth determination.** The depth at each point is obtained from the motion parameters, the image displacement, and the camera parameters.

In parallelizing the Dutta procedure, the key components are the selection within uniform regions, a Hough transform, and array computations. The images used as input for the Dutta depth from motion procedure are shown in Figure 4.5.

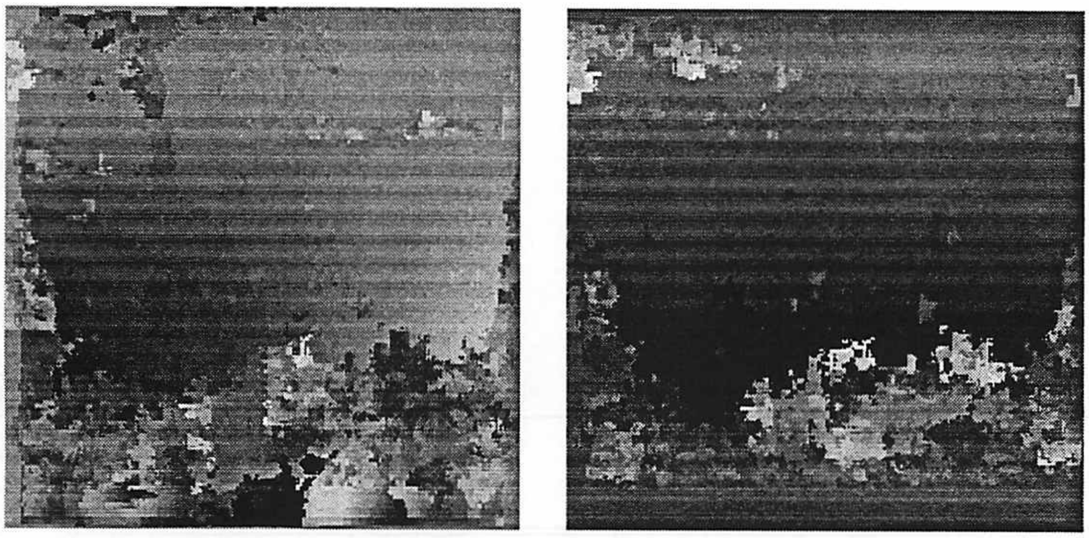


Figure 4.5. Two 256x256 floating point displacement images produced as output by the correspondence matcher and used as input for the Dutta depth from motion procedure. The left image is the X displacement and right image is the Y diplacement.

4.5 Chapter Summary

In this chapter we have examined applications for which MPAs have been found to be particularly cost-effective platforms. From these applications we derived a set of key characteristics, which were then used to select a set of benchmark programs.

The test suite programs can be broadly categorized as follows. See Table 4.3 for a comparison.

- The execution of the curve-fitting filter and the depth from motion procedure is likely to be dominated by floating point computation.
- The execution of the two line finders and the region merge algorithm is likely to be dominated by the communication operations required for region characterization. The Fast Line Finder and the region merger also do substantial arithmetic computation.
- The execution of the correspondence finder is likely to be dominated by nearest-neighbor communication operations and 8 and 16 bit arithmetic.
- The ARPA IU benchmark is (intentionally) composed of a diverse set of tasks, including non-uniform sparse communication, border following, and region characterization.

We will see more precisely what the characteristics are of these programs when the evaluation of the particular architectural components is examined.

Application Code	Description	Characteristics	Data Dependent Execution
ARPA IU Benchmark II	Combines several low- and intermediate-level vision tasks	Integrated series of tasks. 8-bit data computations. Region-based and window-based communication.	Yes
Region Merging Segmentation System	Iterative region characterization	Region-based reductions and broadcasts.	Yes
Daumueller Line Finder	Region-based edge grouping	Region-based reductions and broadcasts.	Yes
Fast Line Finder	Region-based edge grouping	Integer computations. Region-based reductions and broadcasts.	Yes
Correspondence Matcher	Correlation-based correspondence	Integer computations. Window-based communication.	No
Dutta Motion System	Depth from correspondence	Floating point computations.	Yes
Weymouth-Overton Image Preprocessor	Curve fitting filter	Floating point computations. Window-based communication.	Yes

Table 4.3. List of test suite tasks with description and characteristics.

PART III

EVALUATING MASSIVELY PARALLEL ARRAY ARCHITECTURES

CHAPTER 5

ENPASSANT: HIGH-LEVEL OVERVIEW

Recall that the requirements for our evaluation system are that 1) it enable us to systematically search the MPA design space; 2) it therefore needs to be flexible, which implies a simulation-based system; and 3) the test codes must be real programs representative of the target workload. This chapter describes what the evaluation system must look like, given these requirements.

We begin by describing an ideal evaluation system and how it partitions the architecture space into parameters and features. Some of the problems inherent in MPA evaluation—programmability of the test suite and computational intractability of the simulations—are then discussed, together with some standard solutions. The virtual machine methodology, an alternate solution, is introduced and the ENPASSANT system architecture is described.

5.1 An Ideal Evaluation System

An ideal evaluation system would be like the one shown in Figure 5.1. It would have a console where the operator could adjust parameters (by turning dials) and add or remove features (by flipping switches). There would be an input for test codes, and an output where performance results—at the level of clock cycles—would emerge. Moreover, the system would be fast, easy to use, accurate, and flexible.

The distinction between dials and switches is a useful one: for the purposes of evaluation, it makes sense to partition the architectural design space into parameters and features. The parameters denote options that can be varied more or less continuously, such as the width of the ALU and the size of the register file. The features refer to options that are added or subtracted in their entirety. For example, the MasPar MP1 permutation router is a feature. Note that adding a feature may introduce new parameters, for example the datapath width of the permutation router. Thus we see that parameters and features do not cleanly separate into disjoint sets.

Figure 5.2 depicts schematically the current feature space of the family of MPAs. The core set of features are those characteristics that are common to all members of the MPA class. The optional features are those that are present in some, but not all members of the MPA class.

5.2 Problems Inherent in MPA Evaluation

We have already shown that the class of MPAs has not undergone systematic evaluation; thus building a system like the ideal one shown above is probably a useful endeavor. The question is, why haven't such systems been built before?

There are two basic problems: the programmability of the test suite, and the computational intractability of the simulations. Programming issues will be discussed first.

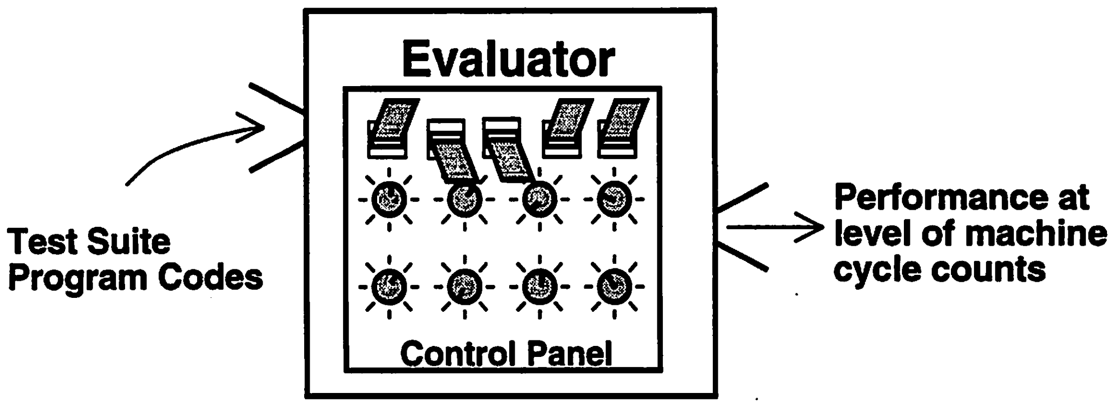


Figure 5.1. An ideal evaluation system would resemble a black box with a console and be fast, easy to use, accurate, and flexible.

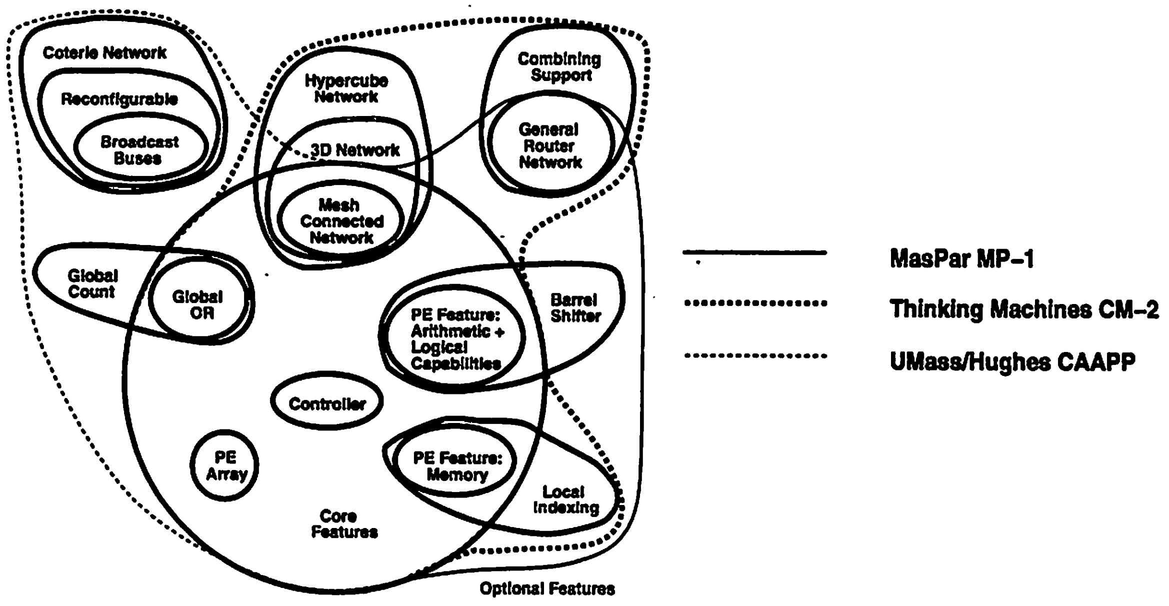


Figure 5.2. Shown are representations of three massively parallel arrays (MPAs) as collections of core and optional architectural features.

5.2.1 Test Suite Issues

The programmability issues revolve around trade-offs between the 'level' of the programmer's model, i.e. how closely it is related to the target machines, and whether the programs are portable among the architectures that constitute the MPA class. The portability issue itself consists of two sub-issues:

1. that of getting programs written for one machine to run on another, what we shall call the *portable language* problem, and

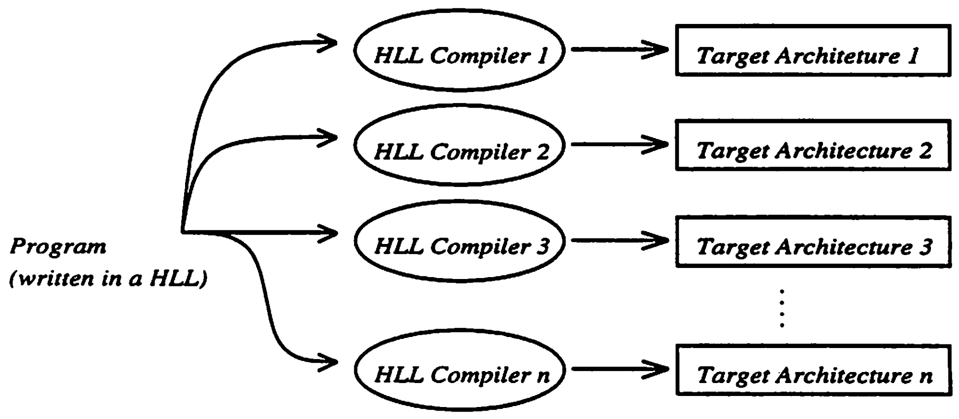
2. a deeper issue that has to do with maintaining fairness in evaluation without requiring excessive amounts of coding, what we shall call the *type architecture* problem (for reasons that will be presented below).

The portable language problem is that of guaranteeing that programs written for the MPA class in general (i.e. the MPA programmer's model) can be transformed into an executable code for any particular member of the class. A different way of formulating the problem is that programs written for one machine in the MPA class should be executable on any other machine in the class.

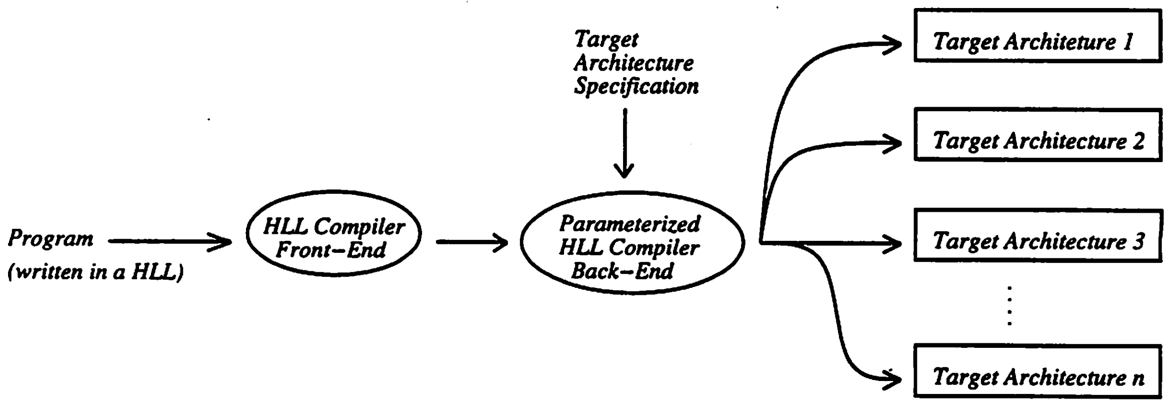
One way to guarantee program portability is to create a language with a compiler for every member architecture, as shown in Figure 5.3a. This is largely how portability is achieved in the class of sequential architectures for languages such as C and Fortran. And because of the large degree of similarity among the members of the MPA class, the MPA compilers for the data-parallel versions of these languages are likely to resemble each other closely. In fact, it is likely that much of the compiler code, perhaps even the entire front end could be reused. A system where, not only the front-end, but also most of the back-end code can be reused is shown in Figure 5.3b. In this scenario, there is a single parameterized back end that generates appropriate code given architectural context. All of these methods of providing program portability for MPAs are likely to require large system building efforts.

The type architecture problem is a consequence of an issue we have already touched on: that for any particular computation task, the presence or absence of a feature is likely to cause a change in which algorithm is optimal for that task. Snyder discusses this issue in [130] where he coins the term *Type Architecture* to refer to any family of machines where, for any given task, the same algorithm is always optimal for all machines in that class. We now discuss this issue in more detail with respect to the two common approaches to test suite design: code oriented and task oriented.

A code oriented benchmark is simply a set of program codes. To use them in architectural evaluation, they are simply compiled and run on the target architecture. This is the approach used by the SPEC benchmark [135]. Code oriented benchmarks work well when two conditions are met. First, each target architecture must have an efficient compiler. Second, the architectures must be similar enough so that the algorithms used to perform a particular task do not favor one architecture over another.



a)



b)

Figure 5.3. Shown are two portable High-Level Language schemes: a) A completely separate compiler for each target architecture, b) the compiler front-end is reused.

These conditions are met for von Neumann uniprocessors, so the code oriented benchmarks work well there. But with MPAs, neither condition is met. Currently there does not exist a language for which there are compilers for every popular MPA. But more importantly, the MPA architectures are not similar enough for the second condition to hold. A code oriented benchmark is likely to force an inappropriate algorithm on some target architecture, thereby skewing the results.

For example, a benchmark for MPAs might include a sorting task. If it is a code oriented benchmark, a particular algorithm will have been used, perhaps a mesh sort. If only meshes are being evaluated, then there is no problem. But if one of the target machines is a hypercube, then the comparison will be unfair as the mesh sort will not give nearly the performance as a hypercube sort.

This shortcoming in code oriented benchmarks is well known, so other benchmarks have been developed that are task oriented. Typically, a set of tasks with very specific required outputs is specified. It is then up to the people who want to evaluate a particular machine to write the test suite code. Benchmarks that are task oriented are the DARPA IU benchmarks [124, 148] and the NAS benchmark [13]. The advantage of task oriented benchmarks is that they are fair, modulo the relative quality of the programming efforts. The disadvantage is that programming the tasks can require programmer-years of effort.

5.2.2 The Performance Issue

Another problem inherent in MPA evaluation is simply the huge computational requirement of simulating such a complex device with enough accuracy to make meaningful architectural decisions. One factor is the large number of PEs: to run a detailed (target machine code) simulation of an MPA requires that the host computer update the state of perhaps hundreds of thousands of PEs for every target machine cycle. Depending on the match between the host and the PEs, each update can require few or many host machine instructions. Another factor is the complexity of some MPA interconnection networks. For example, the CM-2 packet switched router network has similar hardware complexity to the processor array itself; it also has comparable simulation cost. As a result, running a modest sized program that takes milliseconds to run on a target machine can take an impractical amount of time to simulate.

5.3 The Virtual Machine Methodology

To recap the three major problems:

1. There does not exist a portable MPA language. In fact very few languages exist that are supported by even two MPAs. And there certainly does not exist a language with a parameterized compiler back-end (as in Figure 5.3b) to allow us to adjust parameters within a family of machines.

2. MPAs are too varied to make exclusively code oriented benchmarks an option.

3. MPAs are too complex to simulate in detail if we want to explore a significant part of the design space.

We address these problems, in part, by using *virtual machine emulation.*

The basic idea is to simulate a generic MPA at a relatively high level (a virtual machine), generate a trace, and then evaluate the trace with respect to a specific architectural specification.

The virtual machine emulation approach thus allows us to 1) write all our programs in a single language, 2) use a single compiler, and 3) run the simulations much more quickly. In the next sections we show the details of how the virtual machine methodology is implemented and quantify its benefits.

5.4 ENPASSANT Architecture

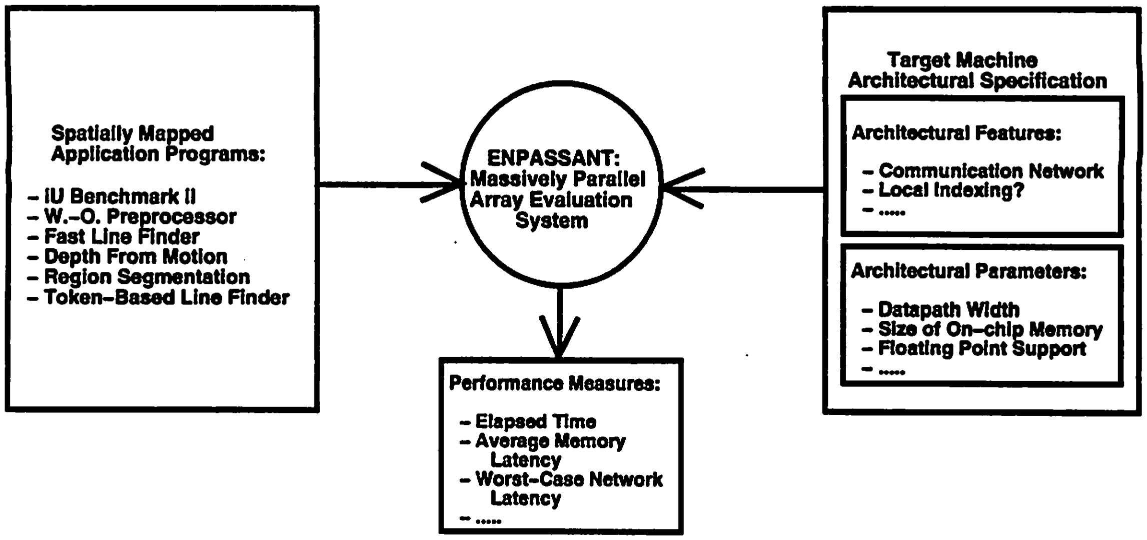


Figure 5.4. ENPASSANT system architecture: highest level view.

We now present an overview of the system architecture for ENPASSANT (ENvironment for PArallel System Simulation ANalysis Tools), a framework for making architectural decisions about massively parallel arrays. At the highest level, ENPASSANT is a black box that takes as input application programs and an architectural specification and outputs performance measures (see Figure 5.4). At a slightly lower level, ENPASSANT contains four major components: the input constructor, the performance model constructor, the virtual machine simulator and trace generator, and the trace analyzer (see Figure 5.5).

- The **input constructor** takes as input application programs written in the IUA class library extension (ICL) and outputs code executable by the virtual machine simulator.

- The **model constructor** transforms the input architecture parameters into instruction, memory, and communication models for use by the trace analyzer.
- The **virtual machine emulator** runs the virtual machine code, and generates execution traces.
- The **trace analyzer** inputs the virtual machine traces and the target machine models and outputs performance measures.

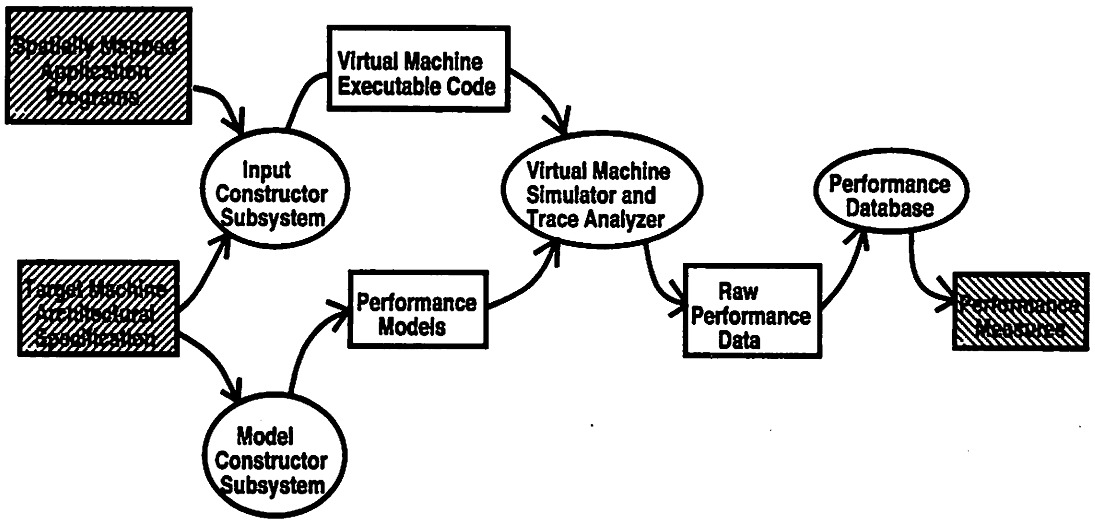

Spatially Mapped Application Programs
Virtual Machine Executable Code
Input Constructor Subsystem
Virtual Machine Simulator and Trace Analyzer
Performance Database
Target Machine Architectural Specification
Performance Models
Raw Performance Data
Performance Measures
Model Constructor Subsystem

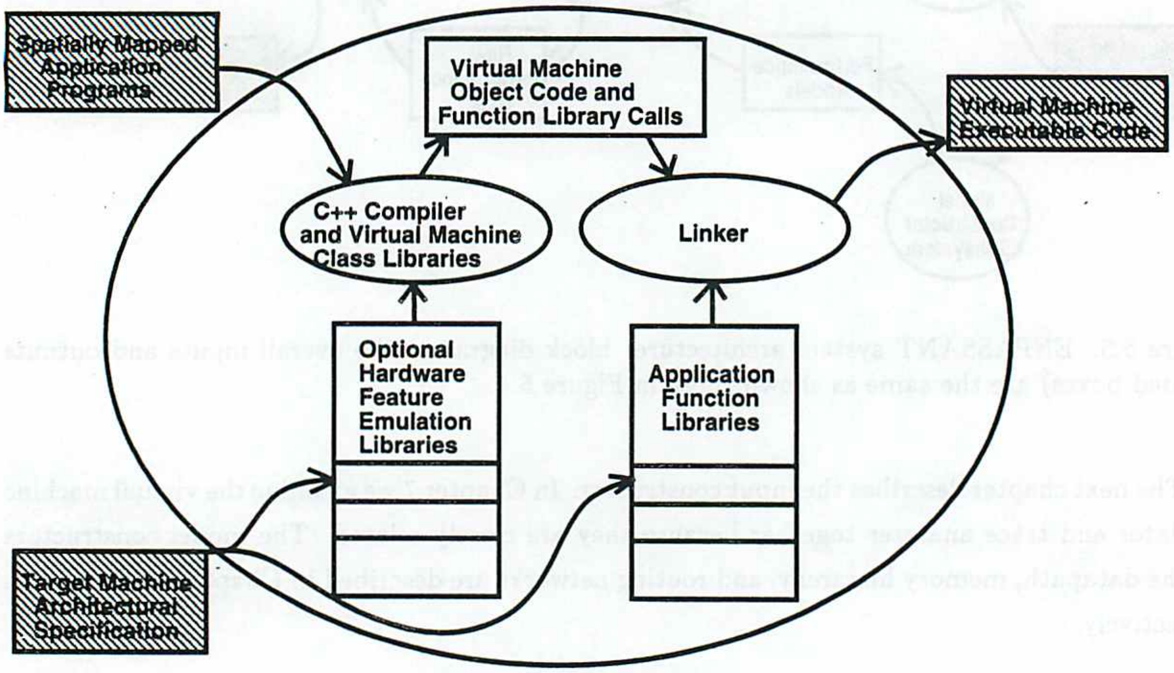
Figure 5.5. ENPASSANT system architecture: block diagram. The overall inputs and outputs (shaded boxes) are the same as shown those in Figure 5.4.

The next chapter describes the input constructor. In Chapter 7 we examine the virtual machine emulator and trace analyzer together because they are closely related. The model constructors for the datapath, memory hierarchy, and routing networks are described in Chapters 8, 9, and 10, respectively.

Chapter 6
Input Constructor Subsystem

6.1 Overview

The role of the input constructor subsystem (ICS) is to deal with the portability issues outlined in the last chapter and to provide the input for the MPA virtual machine. An overview of the ICS is shown in Figure 6.1.



Spatially Mapped Application Programs
Virtual Machine Object Code and Function Library Calls
Virtual Machine Executable Code
C++ Compiler and Virtual Machine Class Libraries
Linker
Optional Hardware Feature Emulation Libraries
Application Function Libraries
Target Machine Architectural Specification

Figure 6.1. Input Constructor Subsystem

The input programs are written in a version of ICL, a language created by augmenting C++ with a parallel class library. The result is a language with similar semantics to MPL and C* and most other data parallel languages. As in any system, the code is compiled and linked to produce an executable form.

There are two other significant features to the input constructor: the emulation libraries for optional hardware features and the application function libraries. The emulation libraries are necessary to support language constructs that are executable directly in hardware by some, but not all, machines in the class of massively parallel arrays. Also, there are some functions within our test suite for which different algorithms are preferable, depending on the routing network. For fairness, these must be provided and are, in the application function libraries.

In the rest of this section we present

- the virtual machine model for the MPA,
- a language that matches this model,
- how the language gets mapped to MPA instances using the optional hardware emulation library,
- how differences in type architecture are dealt with by using selective recoding and application function libraries.

6.2 Selecting an MPA Virtual Machine Model

The goal of the input constructor subsystem is to ensure that the test suite programs all run with maximum possible efficiency on any architecture within the class of MPAs. As we mentioned previously, there are two issues inherent in this process which we refer to as the portable language and the type architecture problems. Both of these issues are affected by the selection of the test suite language, which in turn depends on how we specify the *virtual machine model* for the MPAs in our study.

The virtual machine model (VMM), also called the *Type Architecture* by Snyder [130] and the *Bridging Model* by Valiant [143] is an essential component in both architecture and language design. In the construction of serial languages and architectures, this VMM is the von Neuman model of computation, so familiar at this point that we usually do not even realize that we are using it. But we do: architects (usually) do not design machines with the purpose of running a specific language, say Modula, and language designers (usually) do not create languages to run on specific processors, say the M68040. Rather, there is an implicit intermediate model that is used as a reference point in both endeavors. Both Valiant and Snyder claim that the lack of an analogous, universally accepted model, for parallel processors is the major problem in the acceptance and growth of parallel computation.

The difficulty in selecting a VMM for a class of architectures is that it must simultaneously satisfy several constraints. On the one hand, the VMM must make visible those architectural features that the programmer should be aware of to use the architecture the way it was intended. For example, if there is direct hardware support for a particular operation, say scan, then that operation should be visible in the VMM. On the other hand, the VMM must hide those features whose existence does not affect the way a programmer would write code. For example, the size of the cache is not generally a consideration when writing code, so it is not part of the von Neuman VMM.

Deciding what to include in a VMM involves trade-offs. For example, for ultimate ease of portability and programmability, a VMM that is completely independent of any particular architecture is appropriate. The drawback, however, is that hiding too much of the underlying architecture can cause the programmer to use sub-optimal algorithms. As Snyder shows, an algorithm that a programmer would write for a PRAM may be completely different than the one the programmer would use knowing the underlying PE interconnection pattern of the particular target architecture [130]. It is easy to see how this can lead to terrible inefficiency.

The other extreme is to try for maximum efficiency. In this case, the preferred VMM contains all hardware features that are likely to cause a change in algorithm selection for any particular task. Thus if the choice of parallel sorting algorithm depends on the routing network and an improper choice leads to great inefficiency, then the routing network must be specified as part of the VMM. The obvious drawback of trying for maximum efficiency, is that almost every change in routing network will require a new VMM, with the result that programs will not be portable across MPAs.

One of the first VMM's for MPAs was described by Siegel [127] and consists of a control unit, N processing elements, and an interconnection network. The controller broadcasts the instructions, which are conditionally executed by the processing elements depending on the settings of their activity bits. The choice of interconnection network is left as an issue to be resolved by the designer of the specific machine.

A more general MPA VMM is the *Data Parallel* model of computation [74]. This model is similar to Siegel's, with two exceptions. The first is that instead of assuming N PEs, an unlimited (or constant but unbounded) number is assumed. The second difference is that interPE communication is assumed to require only unit time. This assumption is taken a step further by Blelloch who argues that if memory references over an interconnection network are counted as unit time operations, then parallel prefix (scan) operations—which are of similar complexity—should also be counted as unit time operations [26]. The problem with using the data parallel model as our VMM is that it hides too much of the target architectures from the application programmer and can thus cause great inefficiency. For example, the data parallel model does not distinguish between nearby and arbitrary communication, even though these are likely to have orders of magnitude performance differences.

The MPA VMM we use distinguishes among more types of communications than the data parallel VMM suggested by Hillis and Steele, but does not change with every network as does Siegel's. It has the following characteristics:

- A controller which broadcasts instructions and global data to the PE array.
- An array with a number of PEs as large as the data set.
- PEs with unlimited virtual memory, ALUs that support all conventional operations, but no individual control.

- Feedback from the array to the controller in the form of OR or COUNT. If COUNT is available, it is assumed to be substantially (at least an order of magnitude) more time-consuming than the OR.

- PEs are connected with a routing network that supports arbitrary communication. However, certain routing operations are distinguished. These include mesh moves, broadcast and reduction in contiguous regions, and scans along rows and columns. See the language specification in Section 6.3 for details.

One of the primary criteria in deciding whether a feature can be hidden within our MPA VMM (or MVMM) is that a compiler can be expected to deal with that feature efficiently. For example, a compiler can handle virtual processor emulation; therefore the number of PEs in the array need not be specified. Similarly the compiler can perform PE register and memory allocation, therefore the PE memory hierarchy need not be specified.

One benefit of the MVMM is that it (like the data parallel VMM) is based on the union of the hardware characteristics of the particular machines that constitute the class of MPAs. Thus by using a language based on the MVMM, a programmer will be able to write code that is portable within MPAs.

Another benefit is that (*unlike* the data parallel VMM) the important physical characteristics of MPAs are built in. That is, the programmer is encouraged to use the least expensive operation available for a given task. For example, the mesh-move operator is used whenever possible, rather than simply always using the same general communication operator.

The fact that no MPA in existence has all the characteristics of the MVMM is not necessarily a drawback. For any particular MPA, the individual capabilities of the MPA VMM are either supported directly in hardware, or can be emulated with some level of slow-down. These emulations are discussed at length below.

However, it is precisely the fact that the level of slowdown is not always acceptable that leads to the issue of algorithm portability. This issue is not resolved by the selection of the MVMM. That too is discussed below.

6.3 An MPA Programming Language

Once we have selected an MPA virtual machine model, the language (ICL) follows almost immediately. The name is derived from the fact that the language is C++ with a parallel class library that was originally targeted for the IUA (IUA Class Library) [39]. The major difference between ICL and other data parallel languages, such as C* [138] and MPL [104] is that the parallel data type, the Plane is constrained to two dimensions. Otherwise the features of the VMM are supported. We now briefly describe some of the operations supported by ICL. See Tables 6.1 and 6.3 on Page 69 for a summary. ICL types and methods are denoted with sans serif type style.

- **The Plane Data Type** Planes are declared like any other variable. The size of the two dimensions must be specified as part of the declaration. The semantics are those of a two dimensional array.

FloatPlane
IntPlane
ShortPlane
UIntPlane
UShortPlane
CharPlane
UCharPlane
BitPlane

Table 6.1. Plane data types supported by the ICL programming language

- **Operations on Planes** The standard arithmetic and logical operations are supported on planes. The semantics are the equivalent of performing the operations element by element on arrays. Some new operations are also defined that are useful for operating on bits within elements of a Plane.

- **Activity** The user defined class designator Select is used to specify the Plane elements to take part in a given operation.

- **Feedback** The Any method returns the logical OR of the elements of the Plane specified in the method. Count returns the number of elements set to True.

- **Mesh Routing** The mesh routing operators have the effect of sliding Planes a specified number of elements in a given direction.

- **Permutation Routing, Reductions, and Scans** The routing methods move data according to the addresses specified. If more than one datum is sent to the same destination, then they are combined using the operator specified. Scans are also supported.

- **Region Operations** Region operations are central to spatially mapped computations. Regions are defined as any contiguous set of virtual processing elements or positions within a Plane. Some autonomy is allowed while processing regions in parallel. For example, PEs within regions can broadcast data to those and only those other elements within their region. PEs with certain distinctive values (maximum or minimum value of a Plane element) within a region can be selected.

6.4 Emulating Optional Features

In any data parallel language, including ICL, there are constructs that are supported directly in hardware by some, but not all, machines in the class of MPAs. For the processors that are

Type of Operation	ICL Operator or Method
Arithmetic Operations on Planes	+,-,*,/,%,&,|,¡>>,<<,¬=,+ =,etc.
Logical Operations on Planes	==, =,<=,>=,<,>
Bit Operations	Insert,Bit
Activity	Select active(BitPlane X)
Feedback	Any,Count
Mesh Routing	North,South,East,West
Permutation routing	Route
Reduction routing	RouteOP
Scans	ScanOP
Region Specification	Coterie(CharPlane X)
Region Communication	RegionBroadcast,RegionSelectMin RegionSelectMax,RegRouteOP

Table 6.2. Operations supported in the augmented ICL programming language. C++ methods are shown in sans serif type face

'missing' hardware, these constructs must be emulated using hardware that *is* available; otherwise programs using those constructs will not be portable.

For example, the ICL Route operation can be implemented directly on the CM-2 and the MP1 by using dedicated router networks of those machines. But since Route is not implemented directly on a mesh, it must be emulated using hardware that is available, namely by using the nearest-neighbor communication network [70]. On the other hand, neither the CM-2 nor the MP1 directly support region broadcast, so these machines must emulate that operation. But region broadcast is precisely what the coterie network on the CAAPP was designed to do, so the CAAPP executes that instruction directly.

This requirement for emulation is not unusual: if code containing floating point instructions is run on a machine without a floating point processor, then those instructions are emulated using the integer processing unit. The key ideas are that emulations must be included in a manner transparent to the programmer, and that for fairness, the emulations should be as efficient as possible.

The contents of the Emulation Libraries for Optional Hardware Features are shown in Table 6.4.

6.5 Selective Recoding: Application Function Libraries

To repeat the main problem addressed in the preceding section, it is to create comparably efficient code for each possible machine model within the MPA class. We have just dealt with the question of a portable MPA language, thus answering the question of the mechanism by which this can be accomplished. There still remains the type architecture problem.

It is beyond the capability of the current generation of compilers to recognize that an *algorithm* is inefficient for a given target architecture, much less select or create an appropriate new one. We

Target Machine Network	Constructs Needing Emulation	References
Mesh	Route, RouteOP, ScanOP, Region Operations	[70]
Mesh + Permutation Routing Network	RouteOP, ScanOP, Region Operations	[70]
Mesh + Combining Routing Network	Region Operations	[98]
Mesh + Reconfigurable Broadcast Mesh	Route, RouteOP, ScanOP	[72, 70, 71]

Table 6.3. When a network does not support a construct directly, it must be emulated. Those emulations not referenced are straightforward.

address this problem as follows: sections of the test programs that require different algorithms for efficiency reasons will have them provided. We call this approach ***selective recoding.***

As an example, we look at the task of labeling connected components. The code used on the CM-2 uses either pointer-jumping or segmented-grid-scan based algorithms [98], while that on the CAAPP uses multi-associative leader election via region broadcast [72]. Exchanging or otherwise choosing the inappropriate codes results in both architectures having far worse performance than they are capable of achieving, thus skewing the entire evaluation. The use of the correct algorithm is critical in making fair architectural comparisons. The Application Function Library (AFL) contains the various versions of the critical functions that appear in the test suite.

The following experiment demonstrates the necessity of using the correct algorithm for each machine model. We ran two different connected components functions on ENPASSANT. One was based on the Connection Machine algorithm, the other on the CAAPP algorithm. The performance of both traces was then measured with respect to both CAAPP-like and CM2-like machine models. The slowdown resulting from using the incorrect algorithm on the CM2-like model was a factor of 13.5. The slowdown on the CAAPP-like model was a factor of 81.

It may seem that the number of tasks in the AFL should be the product of the feature space with the task space. However, the actual number is far fewer because many architectural features only require distinct algorithms for a few tasks. These tasks are, in general, those where global communication dominates. Even here, the same code is often optimal (though not equally efficient!) across routing networks. For example, the critical task of summing pixels in regions during a segmentation algorithm simply uses the global +Reduce function (and its emulations) for most architectures.

6.6 Summary

The purpose of the input constructor subsystem is to transform application programs into codes that are executable on the MPA virtual machine emulator (described in the next chapter). In order to implement the input constructor subsystem in a way that it is useful for evaluation, we must deal with portability and fairness issues. The basis of how we do this is the MPA virtual machine model.

The difficulty in selecting an MPA virtual machine is that it be neither too specific, nor too general. Too specific and possible target machine designs are excluded; too general and the domain becomes unwieldy: the operator emulation and application function libraries become too complex.

We have created a data parallel language that maps to the MPA virtual machine and can therefore be used to efficiently program spatially mapped applications for any target architecture within the class. This was done by extending ICL to support hardware features that are available on any MPA instance within our domain (e.g. by adding a family of Scan instructions). In order for programs that use these language constructs to be executable on machines where the constructs are not supported directly in hardware—a necessary condition for an evaluation environment—emulation libraries have been written.

Executing a code as efficiently as possible does not necessarily mean that the code itself is as efficient as it could be. Algorithm selection is sometimes highly architecture dependent. For these cases, various versions of certain tasks have been provided in the application function libraries.

Chapter 7

Virtual Machine Emulator and Trace Analyzer

In this chapter we deal with the problem of how to obtain accurate simulations in a reasonable amount of time. This work is done in the context of the virtual machine emulator and trace analyzer.

7.1 Problems with Trace-Driven Simulation

One of the standard methods used for evaluation in serial and multi-computer architecture research is trace-driven simulation. Since trace-driven simulation also provides the basis for the evaluation technique used in this work, we now present it in more detail.

As we mentioned earlier, the advantage of trace-driven simulation over complete machine simulation is that once a simulation has been run and a trace captured, that trace can be used over and over as different design alternatives are evaluated. The key to this reuse is to simulate the target architecture at a slightly higher level than would be done in a complete simulation. In particular, only the machine-level instruction and register architectures are simulated initially, not those components whose performance depends on instruction ordering such as the datapath pipeline or the memory hierarchy. Those latter components are evaluated with their own simulators which take the trace as input. In this way, any changes in the datapath or the memory hierarchy that do not change the machine-level instruction set can be evaluated without the need to generate a new trace.

The amount of allowable change is limited, however: architectural changes that are substantial enough to require a new compilation, e.g. changes to the instructions set or the register file design, also require that the test program be rerun and the trace generated again. And perhaps more importantly, being able to deal with such architectural changes requires the existence of compiler support for all those architectural alternatives. This support can either be in the form of separate compilers for each architecture, or a flexible parameterized compilation system.

One advantage of trace-driven simulation is the ability to reuse the initial simulation. Another is that the machine-level instruction simulation is substantially simpler than complete simulation, and can thus be expected to run much faster. Yet another advantage of trace-driven simulation is that it can be used to evaluate design modifications of existing machines, again as long as the machine-level instruction set and register file design need not be changed.

An architectural evaluation system based on trace-driven simulation is shown in Figure 7.1. An application program is compiled into target machine executable code, and sent to the machine-level instruction simulator or prototype. If a simulator is being used and if it is flexible enough to support different designs, say changes to the register file, then those architectural parameters must be input to the simulator as well. The program is run and instruction and virtual memory reference traces are captured. The memory reference trace is run through a cache simulator, and the results (cache misses) integrated into the instruction trace. The instruction trace is then run through the datapath simulator.

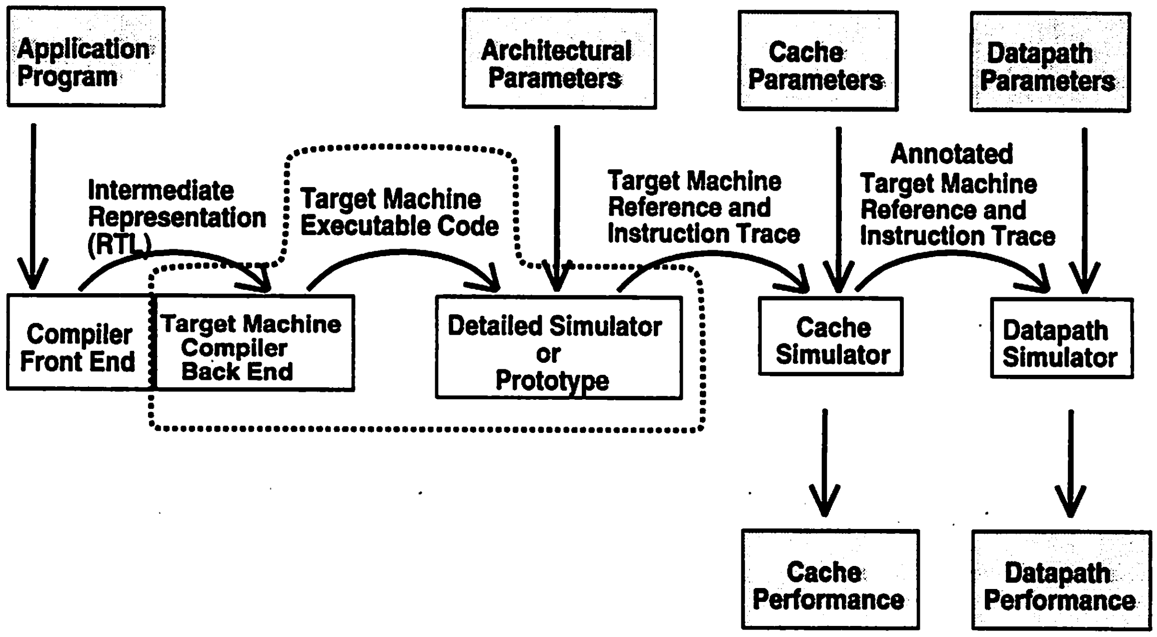

Application Program
Architectural Parameters
Cache Parameters
Datapath Parameters
Intermediate Representation (RTL)
Target Machine Executable Code
Target Machine Reference and Instruction Trace
Annotated Target Machine Reference and Instruction Trace
Compiler Front End
Target Machine Compiler Back End
Detailed Simulator or Prototype
Cache Simulator
Datapath Simulator
Cache Performance
Datapath Performance

Figure 7.1. One view of trace-driven simulation.

An important observation about trace-driven simulation is that, although not all design features need to be evaluated simultaneously (to great advantage), dependencies necessarily exist among components that keep this technique far from ideal. That is, although we would like to be able to evaluate all the components individually and independently from all others, the evaluation of some features depends on the parameters chosen for certain other features. In particular, the cache and the datapath internals must be re-evaluated for every change in instruction set architecture and register file design. And the datapath evaluation itself depends on the cache design parameters chosen. Still, trace-driven simulation is an essential technique for architectural evaluation.

Unfortunately, trace-driven simulation is difficult to transfer directly to MPA evaluation. One difficulty is that the problem of compiler support is even more acute for MPAs than it is for serial architectures. The other problem is perhaps even more critical: machine-level instruction simulation of an MPA is prohibitively expensive on commonly available platforms.

For example, consider the performance of the detailed CAAPP simulator when running on a Sun SPARC-2. A simple program takes minutes to run, while programs of complexity on the order of those in the test suite can take a day or longer to finish. Since turn-around times of this length preclude traversing a significant part of the MPA architectural space, a different method—one that uses the virtual machine methodology—was developed.

7.2 Virtual Machine Emulation and Trace Compilation

The solution to the MPA evaluation performance problem clearly must involve speeding up the machine-level instruction simulation. For convenience, we shall refer to machine-level instruction simulation as a ***detailed simulation***, to contrast it with complete simulation. Figure 1.1 on Page 6 shows the distinction.

Our approach is as follows: just as detailed simulation is coarser grained than complete simulation, so we again increase the granularity of the initial simulation. We refer to this higher-level simulation as ***virtual machine emulation*** after the fact that the method we use is to emulate the MPA virtual machine model (outlined in Chapter 6). In the process of virtual machine emulation a ***virtual machine trace*** is generated.

The MPA virtual machine (MVM) is defined by the comparatively high-level instruction set shown in Tables 7.1 and 7.2. This instruction set follows immediately from the MPA virtual machine model (MVMM). It is in some ways analogous to Paris, the language that Thinking Machines Corporation refers to as the *Connection Machine assembly language*[1]

As will be discussed in the next section, the key behind improving the performance of evaluation systems is the fact that MVM instructions are usually comprised of from 1 to several thousand target machine instructions. By emulating the former rather than simulating the latter, the test program can be run, and a trace generated, in a small fraction of the time necessary to run a detailed simulation.

The cost of this speed up is that the virtual machine trace contains no information about any particular MPA. This information must be inserted later if the trace is to be useful for architectural evaluation. In a way, this gap between virtual machine emulation and detailed simulation is similar to the gap between detailed and complete simulation. In the latter case, the trace obtained from detailed simulation contains no information about the datapath internals or the memory hierarchy. This information is 'recovered' by processing the machine-level instruction trace with respect to the architectural parameters of the target machine datapath and memory hierarchy.

Similarly, the MVM trace is also processed, in this case to 'recover' the target machine-specific machine instructions and the operations on the register file. The basic idea is to run the trace through a series of transformations, wherein information is added with respect to the architectural

[1] *Assembly language* generally refers to a notation that translates one-to-one to machine code. Paris instructions generally translate to a large number of Connection Machine machine instructions.

Virtual Machine Instruction Set - I

Language Construct	**Comment**
Dyadic Operators	
+, − , *, /, %, &, |, ∧, <<, >>	
Monadic Operators	
−, ¬, ++, −−	
Assignment Operators	
=, +=, −=, *=, /=, %=	
&=, |=, <<=, >>=, ¬=	
Convert	cast from one PlaneType to another
Resize	change size of Plane
Conditionals	
==, ¬=, <, <=, >, >=	
Bit Operations	
Insert	move *i* BitPlanes starting at *j* in input to the *i* BitPlanes starting at *k* in output
Bit	return the *i*th BitPlane
Read or Alter PE Status	
Edge, NEdge, SEdge, WEdge, EEdge	Return edge status in BitPlane
RowIndex, ColIndex, Index	Return index in ShortPlane or IntPlane
Coterie	Set regions according to CharPlane input
Activity	Set activity to BitPlane input

Table 7.1. Shown is the part of the MPA virtual machine instruction set that corresponds to operations executed internally within PEs.

Virtual Machine Instruction Set - II

Language Construct	**Comment**
Inter-PE Communication	
North, South, East, West	2D mesh interPE communication
RegionBroadcast	One-to-many transfers, only within regions
RegionSelectOP	Return BitPlane with location of all min or max values in each region set
Route	One-to-one interPE communication
RouteOP	Many-to-one combining interPE communication
RegRouteOP	Same as RouteOP but destination must be within region
ScanOP, ScanRowOP, ScanColOP	Scans and segments scans
Array/Controller Interaction	
Any	Return scalar OR of input BitPlane
Count	Return scalar count of input BitPlane
Access	Return scalar OR of BitPlane i of input
ToArray	Converts a Plane to a scalar array
FromArray	Converts a scalar array to a Plane

Table 7.2. Shown is the part of the MPA virtual machine instruction set that corresponds to interPE communication and array/controller interaction.

parameters of the target machine. We call this process *trace compilation* and discuss it in detail below.

The virtual machine emulation and trace compilation methodology is illustrated in Figure 7.2. As in trace-driven simulation, the test suite program (written in ICL) is input into the compiler. However, the output of the compiler is not target machine executable code, it is rather (at least conceptually) a target machine independent, generic, MPA executable image. The executable code is run on the MVM emulator and an MVM trace is generated. The virtual machine trace is then run through the trace compiler, where functions such as instruction expansion and register allocation and assignment are performed. The output of the trace compiler is the reconstructed machine-level instruction trace for the target machine, which can then be processed in the same way as it would be in trace-driven simulation.

7.3 Virtual Machine Emulation Details

In order to explain the issues involved in implementing MPA virtual machine emulation, it is necessary to first discuss three issues:

- the relationship between the virtual machine and the target machine instructions,

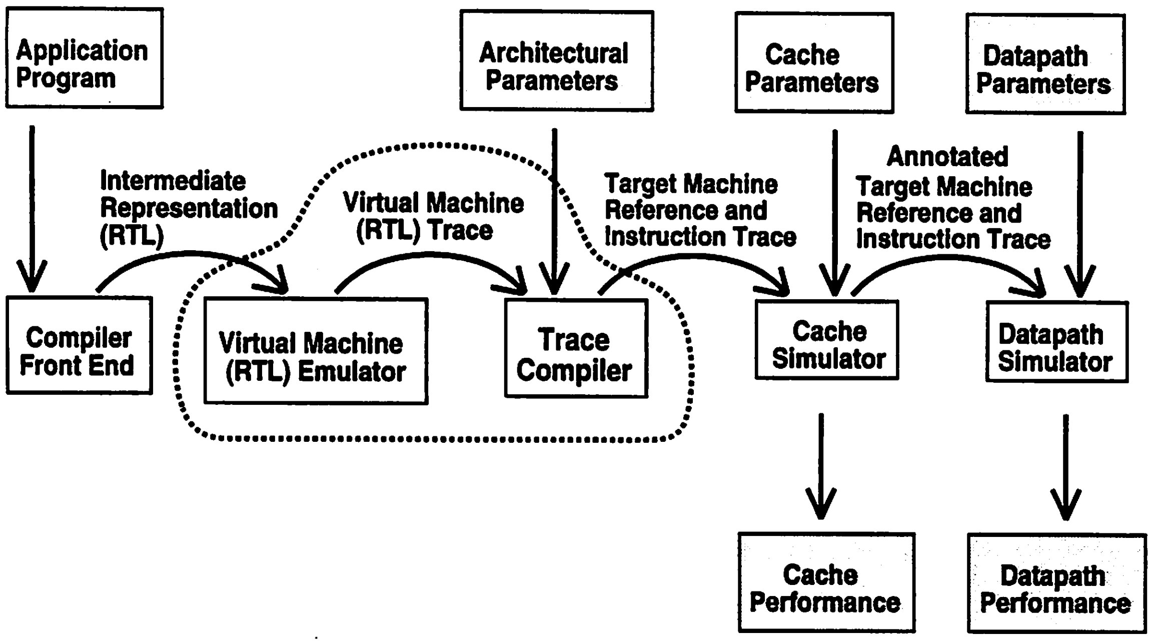

Application Program
Architectural Parameters
Cache Parameters
Datapath Parameters
Intermediate Representation (RTL)
Virtual Machine (RTL) Trace
Target Machine Reference and Instruction Trace
Annotated Target Machine Reference and Instruction Trace
Compiler Front End
Virtual Machine (RTL) Emulator
Trace Compiler
Cache Simulator
Datapath Simulator
Cache Performance
Datapath Performance

Figure 7.2. How to use the virtual machine methodology and trace compilation to evaluate architectures.

- how instructions are issued from the controller to the array, and
- how ICL programs are compiled into target machine code.

The first issue is that a substantial number of the target machine instructions are often necessary to implement any particular MVM instruction. This is because the MPA PEs are generally very simple. For example, an MVM instruction that adds a 32-bit integer to another requires 65 instructions to implement on the first generation CAAPP, even assuming that the operands are already in register file. One instruction is needed to explicitly clear the carry bit, and two instructions per bit are needed to add the two operands. To execute complex MVM instruction such as floating point and routing operations on the CAAPP, thousands of instruction are often needed. Although this number is somewhat smaller for recent MPAs with more complex PEs, the fact remains that the mapping of instructions from target machine onto virtual machine is usually many to one.

The second issue is that the MPA controller is often not monolithic. More often the situation is as in Figure 7.3 where the controller has two parts: one that issues macro instructions of a type similar to the MVM instructions and a second that issues the actual target machine instructions. The mechanism by which this translation occurs can be very complex, but will not be discussed further in this work.

The third issue is that compilation is also not a monolithic process, but is rather composed of phases, such as those shown in Figure 7.1. Typically the front end of a compiler transforms the

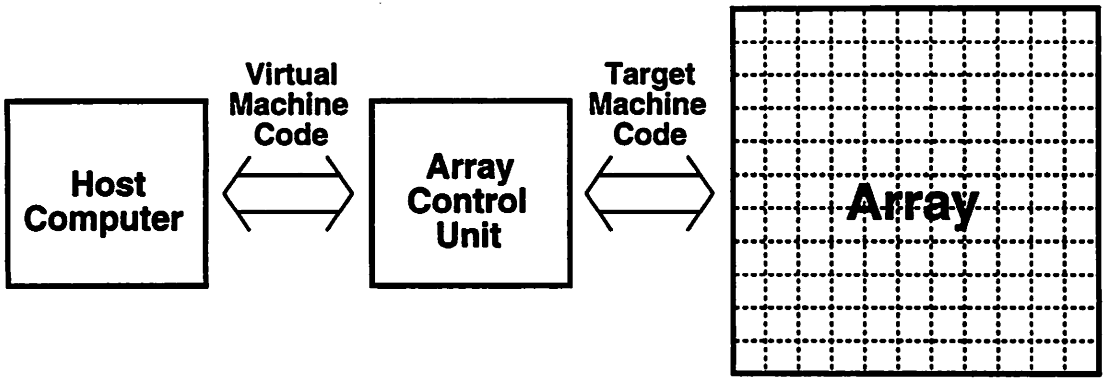


Figure 7.3. Macro-instructions are issued by the host to the controller, which generates and issues the micro-instructions instructions actually executed by the PEs.

input into an intermediate representation such as a register transfer language (RTL). From there, the compiler back end will translate the RTL instructions into the target machine code.

As an example, we consider the two ways that programs written in ICL code can be compiled into CAAPP code. In the first, target machine micro-code is generated within the parallel methods. In the second, the micro-code is generated within a modified compiler from the RTL representation.

There are three points to this discussion:

1. The MVM code is intimately related to the ICL language: the MVM instructions are precisely either the RTL array instructions, or the macros executed when the ICL methods are invoked.

2. The partition of MVM code from target machine code is a natural division, not just from the standpoint of the programming model, but also from that of architectural implementation. This is seen in Figure 7.3. In fact it is quite plausible for the array to be replaced by another, very different, processor that is also capable of executing the MVM instructions.

3. There are at least two ways that MVM emulation could be implemented. The first is to apply the method shown in Figure 7.2 literally: the application code in the RTL representation is emulated on an RTL emulator. This would require building a very complex system, however, as both controller and array instructions would have to be emulated. The compiler would also need to be rebuilt and the code rerun for every change in register file design. The second way is to take advantage of the partition illustrated in point 2: instead of issuing the MVM instructions to the array, let them be executed on a convenient, available machine. That is, let the *host machine itself* execute them. This second method is relatively simple and efficient.

Here are some details of how MVM emulation is implemented. The ICL application codes can be compiled using one of three different versions of the C++ methods. Respectively, they translate the MVM instructions into either: 1) directives to the controller to broadcast the appropriate CAAPP micro-code sequences, 2) Paris code for the Connection Machine, or 3) generic C array operations that can run on any machine on which C is supported. In any of these modes, the issuance of an MVM instruction can be recorded as it is executed providing the mechanism by which the trace is generated for the MVM emulation.

7.4 Evaluating Virtual Machine Emulation

In order to evaluate the Virtual Machine Emulation technique, we need to determine two things: the correctness of the results and the performance.

7.4.1 Correctness

We know that the code generated for the virtual machine emulator matches the code generated for the CAAPP simulator because they both generate the same intermediate representation (RTL). The difference is that one back end generates CAAPP microcode while the other generates code for the host machine. We have confirmed the correctness of this procedure by running test programs on both the CAAPP simulator and the virtual machine emulator and obtaining identical results.

7.4.2 Performance

Virtual machine emulation is also very fast: we have found that virtual machine emulation is a factor of 65 faster than simulating the CAAPP at the instruction level when both are run on a Sun Sparc-2 processor. This speed up comes from a number of sources:

- MVM instructions map very well onto RISC processors: many operations that take up to 2000 cycles on the CAAPP are done in, at most, a few cycles on the SPARC-2.
- RISC processors are optimized to run array operations. They take advantage of the data locality with caching, pipelining and other standard techniques.
- ICL produces relatively high quality serial machine code: running an application code written in ICL results in little slowdown over code written directly in C.
- Because of the simplicity of MPA PEs, MVM instructions do not map as cleanly onto the CAAPP instruction-level simulator as they do onto RISC processors: floating point operations are particularly problematic.
- The slow-down when running a detailed simulation on a serial machine is more than just the difference in how efficiently the host machine and the target machine PEs can execute the MVM instructions. Rather, the slow-down is at least as much the result of the host

simulating an array of PEs that is executing MVM instructions. That is, the host must do more than just execute the equivalent target machine instruction for every PE: it must also keep track of all of their internal states.

7.5 Trace Compiler Overview

A critical part of the virtual machine methodology is to process the virtual machine trace so as to *reconstruct what the behavior of the target machine would have been* had the target machine been running the input code, rather than the virtual machine emulator. This reconstruction is performed by the trace compiler. Essentially, what happens is that the tasks that are normally executed by the compiler back end to create target machine executable code (a step that was by-passed to run the virtual machine emulation) must now be reconstructed by the trace compiler.

One of the tasks normally executed by the back end of an MPA compiler is to generate virtual processor emulation code. This task is not needed on serial machine compilers, but is essential for MPAs. As a rule, there are never enough processors to map data within parallel variables one-to-one onto the processing elements. The standard method of dealing with this issue is for there to always be sufficient *virtual processors* to create the one-to-one mapping. Then the physical processors emulate the required number of virtual processors. Since the code for this emulation is normally generated by a compiler back end, that function must be reconstructed by the trace compiler.

The other tasks that are executed by the trace compiler are standard: allocate registers, assign registers and memory locations, deal with register spilling by generating loads and stores, perform peephole optimization, and generate the target machine code.

The basic flow of the trace compiler is as follows. After the virtual machine trace is generated, it is passed through a series of transformations during which the trace compiler functions are carried out. Some of these functions require information about the target architecture, so this must be added when appropriate. See Figure 7.4 for the basic flow. The details are presented in the next sections.

7.6 Trace Compiler Design Issues

There are several aspects to getting good trace compiler performance: 1) minimizing the total number of transformations to which the trace must be subjected, 2) minimizing the time required for each transformation, and 3) maximizing the number of transformations that can be reused to obtain a particular target machine performance datum.

These factors are inter-related: reducing the total number of transformations is simple if each transformation is made arbitrarily complex. A more realistic issue is balancing the amount of information that must be carried along with the trace against the ability to reuse computation.

The performance goals are also naturally constrained by the requirement that some actions must logically precede others: We need to know which variables must be in registers (allocation)

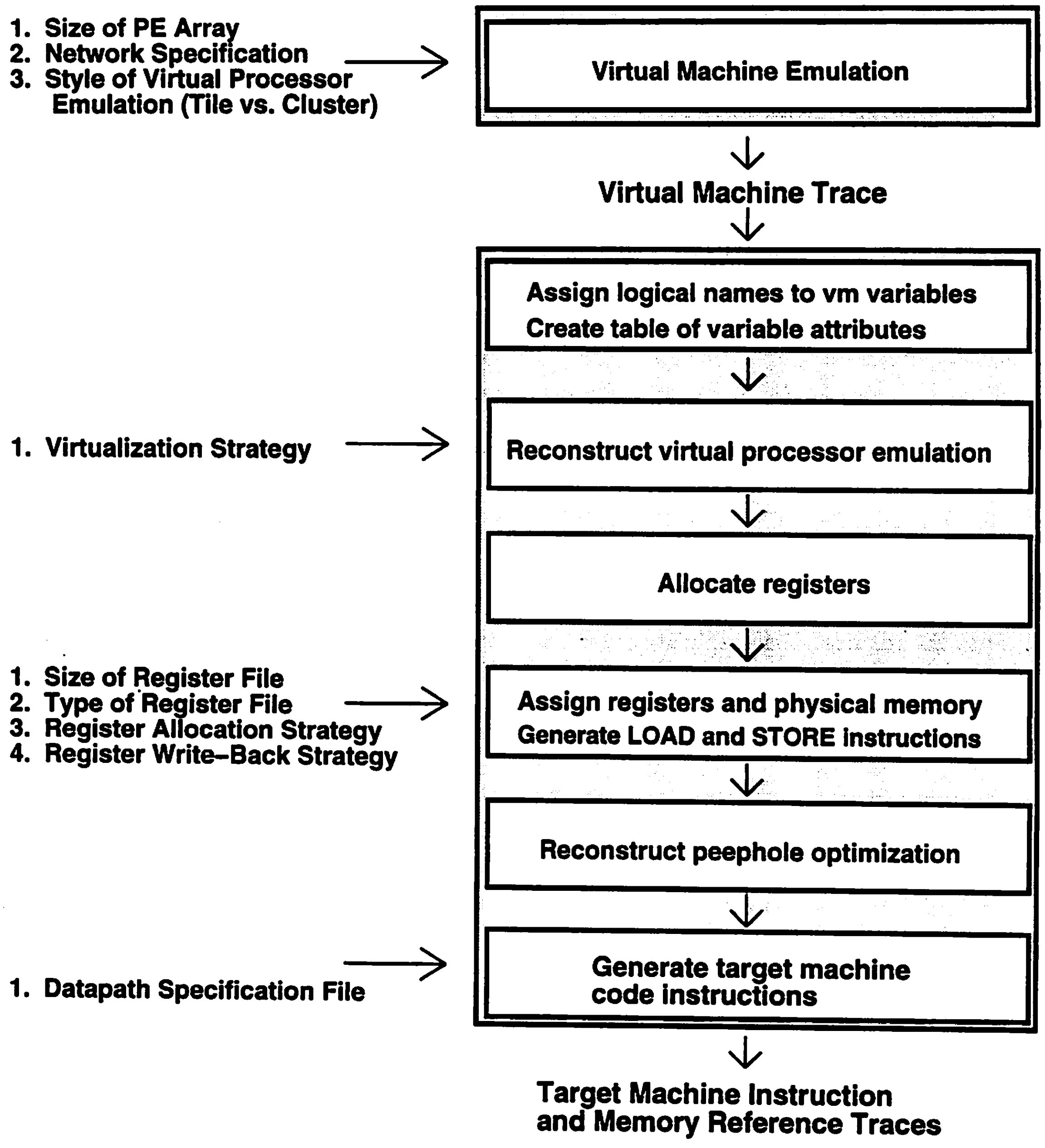

1. Size of PE Array
2. Network Specification
3. Style of Virtual Processor Emulation (Tile vs. Cluster)
Virtual Machine Emulation
Virtual Machine Trace
Assign logical names to vm variables
Create table of variable attributes
1. Virtualization Strategy
Reconstruct virtual processor emulation
Allocate registers
1. Size of Register File
2. Type of Register File
3. Register Allocation Strategy
4. Register Write-Back Strategy
Assign registers and physical memory
Generate LOAD and STORE instructions
Reconstruct peephole optimization
1. Datapath Specification File
Generate target machine code instructions
Target Machine Instruction and Memory Reference Traces

Figure 7.4. The virtual machine trace is passed through a series of transformations to reconstruct the target machine instruction and memory reference traces.

before we can be decide which particular register will be used by which variable (assignment). A further constraint is that some of the trace compiler tasks depend on parameters of the target architecture.

Ideally, we would like to be able to capture and use a single MVM trace for use in evaluating all target architectures. This ideal is certainly possible, but turns out to be suboptimal because of the dependencies involved. Two examples are that 1) communication network performance often depends on the routing pattern and the PE virtualization factor, and 2) just as in trace-driven simulation, cache performance depends on the register file design.

Two ways the network evaluation dependencies can be dealt with are: 1) by evaluating network performance during virtual machine emulation (trace generation) or 2) by saving the routing patterns (e.g. along with the trace) for later evaluation. The problem with the former method is that network simulation is very time consuming, often significantly more so than the rest of the virtual machine emulation. The problem with the latter method is that the context for each routing instruction is on the order of a mega-byte of data, making storage impractical. ENPASSANT supports the first option, as well as a third alternative: the data-dependent routing network and the rest of the target architecture can be evaluated separately.

In the other dependency example, cache design performance depends on register file design because cache behavior is a function of locality within the memory reference trace. The memory reference trace in turn depends on the register file characteristics. Thus, systematic memory hierarchy analysis requires that separate memory reference traces must be generated for each register file specification. The flip side of this example is that most of the effects of varying the register file can be determined independently of the cache architecture.

7.7 Trace Compiler Details

When the virtual machine instructions are captured, they are recorded in the form of records with the following fields:

- the kind of instruction,
- the types of the operands,
- whether the operands are constants (broadcast by the controller) or PLANE variables,
- the virtual machine labels of the plane variables, and
- miscellaneous information that is essential for further evaluation of some instructions.

Recall that the instruction stream consists of array directives broadcast by the controller. Thus there are no flow-of-control instructions (e.g. JUMPs) and no local address autonomy (no index variables). See Tables 7.1 and 7.2 for the virtual machine instruction set.

Trace compilation involves running the trace through a series of transformations, or passes, the details of which are now presented.

7.7.1 *Pass 1 – Create 'Symbol Table'*

Since there is no guarantee that the virtual machine label of a Plane variable is unique, such a unique identifier must be selected. This can be done by tracking virtual machine labels as they are allocated and then deallocated.

Another function of Pass 1 is to obtain variable information that is useful later in the process. Variables are determined to be static, dynamic, or temp. Static variables are recognized by the fact that they are never allocated. Temp variables are recognized by context. The virtualization factor of each variable is also determined at this point.

7.7.2 *Pass 2 – Perform local compiler optimizations*

The first optimization minimizes the amount of computation and number of temporary registers required in statement evaluation. Part of this process involves reconstructing and reducing the statement evaluation trees. This includes 1) combining OPs (+,−,etc.) and SETs (=) into OPSETs and 2) transforming OPs into OPSETs if one of the operands is the same as the result. At the end of this pass, each temporary variable is tagged to indicate the depth of the evaluation stack that it represents, which is used when registers are assigned.

7.7.3 *Pass 3 – Generate virtual processor emulation code*

If an operation is performed on a variable that has a virtualization factor of i, each physical PE must emulate i virtual PEs during the execution of that operation. That means, for each variable with a virtualization factor i, $i-1$ new variables must be created.

If the instruction does not involve interPE communication or array-to-controller feedback, then virtual PE emulation is straightforward. For example, the virtual machine instruction

$$\text{ADD}, A, B, C$$

expands to the instruction sequence

$$\begin{gathered}\text{ADD}, A_1, B_1, C_1\\ \text{ADD}, A_2, B_2, C_2\\ \vdots\\ \text{ADD}, A_i, B_i, C_i\end{gathered}$$

If the instruction involves interPE communication, e.g. nearest neighbor communication, then the virtual processor emulation is more complex than simply replicating instructions. For example, if the virtualization factor is 4, then the virtual machine instruction

$$\text{WEST}, A, B$$

becomes the instruction sequence:

```
ALLOCATE, $T_{next}$
WEST, $A_1$, $B_1$
WEST, $A_2$, $B_2$
WEST, $A_3$, $B_3$
WEST, $A_4$, $B_4$
ACTIVITY, EastEdge
SET, $B_2$, $T_{next}$
SET, $B_1$, $B_2$
SET, $T_{next}$, $B_1$
SET, $B_4$, $T_{next}$
SET, $B_3$, $B_4$
SET, $T_{next}$, $B_3$
ACTIVITY, ActivityStack
DEALLOCATE, $T_{next}$
```

The emulations for the array feedback operations are slightly simpler, but the emulations for Broadcast and RegionSelectOP are substantially more complex. These will be discussed in the chapter on virtual processor emulation.

A significant issue in virtual PE emulation is the number of instructions for which a physical PE should emulate a particular (ith) virtual PE before 'changing context' and emulating the next ($i+1$th) virtual PE. For instructions that span virtual PEs, such as feedback and interPE communication, the issue is moot as all virtual PEs must be emulated together. Other instructions, however, have no interaction among virtual PEs and can therefore be reordered if there are performance reasons to do so. Such instructions are the logical and arithmetic array operations.

It turns out that there *is* a performance reason to reorder those instructions without dependencies across tiles. That is because each virtual PE has its own context and working set: consequently changing emulation from one virtual PE to the next also requires changing the context and the working set. Just as in any other kind of thread management, switching context should be done as infrequently as possible.

The instruction reordering takes place as follows. The virtual instruction trace is examined and sequences of independent (array) instructions are marked as such. These sequences are necessarily bounded by communication and feedback instructions. These 'independent' sequences are precisely the sets of instructions that should be emulated for each virtual PE before the context must be changed and emulation of the next virtual PE begins.

7.7.4 Pass 4 – Generate register allocation directives

Register allocation directives are instructions to the register assignment pass indicating that certain MVM variables must be in a register at a particular time. The allocation requirements are simple given that the assumed target machine PE architecture is a RISC machine where all operands in arithmetic/logical instructions must be registers. For example if the following instruction

```
ADD, $A_3$, $B_3$, $C_3$
```

appears in the MVM trace, then the MVM variables A_3, B_3 and C_3 must have registers allocated for them prior to its execution.

7.7.5 Pass 5 – Memory and Register Assignment

Logical variables are assigned to physical memory locations by allocating space from a heap.[2] The information used in this process includes the Allocate and Deallocate directives and whether or not the variable is static.

Assigning registers, however, is considerably more difficult than assigning memory locations. This is because, although trace compilation is similar to the process we normally refer to as compilation, there are significant differences. On the one hand, much information available in a static program, such as basic block boundaries, is not available in a trace. On the other hand, there is the advantage of having knowledge that the program compiler sometimes does not have, such as the last use of a variable.

As a consequence it is impossible, given only a behavioral trace, to reconstruct precisely the way that registers would have been assigned by a program compiler. What can be done, however, is to *bound* the possible behavior. Given a trace and a register file specification, the register file can be evaluated using the same techniques that would be used to evaluate an explicitly managed cache. Two different policies can be assumed for register assignment: one that is generally better than a compiler could do, the other that is generally worse. An example of the former is LRU allocation. An example of the latter is reserving a number of registers for the evaluation stack and allocating the rest of the register file using random replacement. By using these two policies together, the spread of possible performance for a given register file design can be bounded. See for example Figure 9.5 that these bounds appear to be narrow.

Once the MVM variables have been assigned to physical memory and registers, the generation of Load and Store instructions follows immediately.

7.7.6 Pass 6 – Peephole Optimization

This phase gets rid of obvious inefficiencies such as Stores followed immediately by a Load of the same variable, and Stores to temporary variables that are never used again.

7.7.7 Pass 7 – Target Machine Code Generation

Each virtual machine instruction is expanded into its constituent target machine-level instructions. The details of this pass are given in Chapter 8.

[2] Referring to the memory locations as *virtual* or *relative* is more accurate, but confusing in that virtual already has two meanings in this work.

7.8 Evaluation of Trace Compilation

In order to evaluate the trace compilation technique, we need to determine two things: the performance and the validity of the results.

7.8.1 Trace Compilation Performance

The performance of the trace compiler is comparable to that of the virtual machine emulation that generates the trace initially. To see how this is so, recall that for every instruction in the virtual machine trace, the host performs an array operation. In contradistinction, significantly less processing is required for every record during trace compilation. In fact, most of the time is spent on disk I/O, and on compressing and uncompressing the traces.

7.8.2 Limits of Trace Compilation

Ideally, we would like the virtual machine emulator and trace compiler to generate an exact replica of the machine code trace that would emerge from a prototype, or from the matching detailed simulator. We refer to this trace as a detailed simulation trace. Achieving this ideal, however, is difficult if not impossible: too much information is lost when a program is compiled and run. For example, basic block and procedure boundaries—used for register assignment and code optimization—are not available to the trace compiler.

An exact match is not required, however. It is not necessary for the virtual machine emulator and trace compiler to generate target machine code traces that are identical to the detailed simulation trace. Rather, the traces generated must only be sufficiently similar to be useful for architectural evaluation. We will now demonstrate that this is done by ENPASSANT.

We begin by recalling that the compiler front-ends are identical, regardless of whether detailed simulation or virtual machine emulation are being used. And the correctness of the virtual machine emulation is easily verified by comparing it to an existing detailed simulation. That leaves the compiler back-end to be verified. There are three components in the back end: code generation, register allocation, and code optimization.

7.8.2.1 Code Generation

We verify the correctness of the MVM instruction expansion by asserting that the technique used is that same as that used by the ICL compiler back-end. In particular, a parameterized target machine code generator (described in Chapter 8) is used.

7.8.2.2 Register Assignment

The effect of register assignment on architectural evaluation is easily isolated. The compiler's success in assigning registers results in more or fewer Loads and Stores being required during the execution of a task. Put another way, the quality of the register assignment policy results in a task having (apparently) a larger or smaller working set. A larger register file is therefore needed to compensate for poor register selection.

Thus if the trace compiler assigns registers more (or less) efficiently than the code compiler, it will appear that a smaller (or larger) register file is sufficient (required) than that which is really needed to obtain certain performance. However, as is shown in Section 9.2.1, the effect of drastic changes in register assignment policy make only a small difference in the measured effect of the register file size on the number of Loads and Stores in a trace.

7.8.2.3 Code Optimization

The effect of code optimization on architectural evaluation is to change the proportion and, in some cases, the order of the target machine instructions. For example, a code optimizer for a serial processor will tend to eliminate more arithmetic and load/store instructions than flow-of-control instructions, increasing the proportion of branch instructions executed. A likely effect is to make a shorter datapath pipeline desirable [69]. A serial processor evaluation system should thus include code optimization in order to discover such effects.

There is a similar effect in MPA architectural evaluation. For example, code optimization increases the proportion of feedback instructions (used in flow-of-control). Since feedback instructions delimit virtual PE emulation code segments, the likely consequence of MPA code optimization is a smaller virtual PE working set size. It is therefore desirable to reconstruct compiler code optimizations for MPA evaluation.

Code optimizations are reconstructed in Passes 2 and 6 described above. How they compare to commercial optimizing MPA compiler back-ends is impossible to determine, however, since such compilers have not been made public. Also, the approach of the previous section—where we showed that changes in performance due to register assignment policy could be bounded—does not work here: the consequences are much more wide-spread. What we will show, however, is that the nature of MPA program codes and of the trace compilation process precludes most common optimizations, other than those already reconstructed, from being significant.

We begin by briefly reviewing code optimization, for more details see, e.g. Aho, Sethi, and Ullman [6]. Optimization can be done at all levels: on the input code, on the intermediate code, and on the machine code. The advantage of working on the intermediate code is that many constructs, such as the address computation, have been made explicit and so can be manipulated. The most common optimizations (besides peephole optimization and effective register assignment) can be categorized as function preserving transformations or loop optimizations. Examples of the former are common subexpression elimination, copy propagation, dead code elimination and constant folding. Examples of the latter are code motion, induction variable elimination, and strength reduction.

Three points are important for this discussion. The first is that the most effective code optimizations are those that improve performance inside of loops. The second is that most optimizations arise from improving address calculation: this is especially true of common subexpression elimination and strength reduction. The third point is that many transformations become significant only as a result of previous optimizations. To quote Aho, Sethi, and Ullman on dead code elimination: "While a programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations."

We now examine code optimization in the context of MPA programs. There are two points to be made here. The first point is that MPA programs are composed of two types of instructions: controller instructions that run on conventional serial processors and instructions that are broadcast from the controller to the array. Flow-of-control and scalar index calculations are handled entirely by the controller. The second point is that MPA programs, by their nature, have many fewer loops than serial programs, and those tend to have relatively few iterations. This is because parallel variables are themselves two-dimensional arrays: Single MPA instructions thus typically perform the function of the inner two loops of the comparable serial program.

To put all this together:

The trace compilation technique does not preclude optimizations from being performed by the compiler front end. Thus a pass of common subexpression elimination and dead code removal can be performed there.

The MPA intermediate array code does not contain the same opportunities for optimization that are contained in the comparable serial code. The loops are nested far less deeply, reducing the number of induction variables, and thus the opportunities for strength reduction and induction variable elimination. Taken together, this results in far fewer opportunities for derivative transformations, such as copy propagation and dead code elimination.

Most of the remaining common optimizations are already executed by the trace compiler, in particular: removal of excess temporary variables, statement tree reduction, peep-hole optimization, and efficient register assignment.

7.8.2.4 Summary of Trace Compilation Evaluation

We summarize why we are confident the traces produced by the virtual machine emulator and the trace compiler closely match the traces produced by the equivalent detailed simulator.

- The compiler front ends are identical. Therefore the macro instructions generated for the controller are identical.

- The results produced by the MVM emulator match the results produced by the detailed simulator. Therefore the MVM trace is correct.

- The target machine code is generated by the parameterized datapath model described in Chapter 8. As long as the target machine can be subsumed by this model, the code generation will be correct.

- There *is* room for error in reconstructing register allocation and code optimization. However, we have shown that because of the nature of MPAs, the code is far less amenable to sophisticated transformations than, say, serial machine code, and that the limits of performance can be bounded.

7.9 Conclusions about the Virtual Machine Emulation Methodology

We now summarize the advantages of the virtual machine methodology, especially (in the first two points) with respect to trace-driven simulation. The benefits of the virtual machine and trace compilation methodology are as follows:

- Virtual machine emulation is nearly two orders of magnitude faster than detailed simulation. This is critical as it means not only that more experiments can be run; it is the difference between being able to, and not being able to, do this research,

- Traces only need to be generated relatively rarely, much less frequently than for trace-driven simulation. For example, traces do not need to be regenerated when the register file size changes. If certain target machine characteristics such as the number of processing elements and the type of routing network is known, then each virtual machine trace needs to be generated only once. Since the compilation process is in general much faster than the virtual machine emulation, this is another significant advantage.

- The results are precise: traces are generated at the machine instruction level. This means that evaluation can be done in terms of cycle counts.

- The accuracy is comparable to that of running detailed simulations, say, on the CAAPP simulator.

- ENPASSANT is portable. The primary host system hardware requirements are an adequate amount of memory for the virtual machine emulation and sufficient disk space to hold the traces. The examples found in this dissertation were generated on a SPARC-2 workstation with 96 megabytes of memory and 1.5 gigabytes of disk space. The primary host system software requirements are that it run a flavor of UNIX, X-Windows, and C++.

PART IV

EVALUATING MASSIVELY PARALLEL ARRAY COMPONENTS

CHAPTER 8

DATAPATH EVALUATION

As we stated in the introduction, computer architecture is critically affected by advances in VLSI technology. Especially significant for the purposes of this study is that chip device counts are increasing *quadratically* with time. How MPAs should best take advantage of these extra devices revolves around two issues: how to partition the chip area into PE datapath and memory components, and what datapath designs can best take advantage of the space that has been allocated to it. The latter issue is discussed in this chapter, the former in the next.

One way to use increased chip area to improve datapath performance is to add complete functional units, such as multipliers, barrel shifters, and/or floating point processors. Other uses of the chip area are to increase the area spent on particular components that are already in the processor, for example widening the PE ALUs and datapaths. The ENPASSANT Datapath/ALU evaluator supports all of these design choices.

This chapter is organized as follows. We begin by giving details of the state of the MPA PE datapath design space, emphasizing designs from the last five years. After that, the presentation of the datapath evaluator begins with an overview of the issues involved in datapath evaluation. There follows a description of the basis for the datapath evaluator, the parameterized datapath model. Next comes a detailed description of the components of the datapath evaluator. We end this chapter by presenting some of the implications of datapath design on the complexity of MVM instructions and a series of case studies that show how ENPASSANT can be used to evaluate the effect of changing the ALU width on program execution time.

8.1 The Current MPA Datapath Design Space

We present the current state of the MPA PE datapath design space by describing the datapaths of some typical and well-known MPAs. We restrict our consideration to machines that have actually been built, and that—except for the ILLIAC-IV—are relatively recent.

Some of the terminology differs from processor description to processor description. In particular, in older designs the term *register* is used to refer to any of the small number of one bit accumulators that are used for the ALU input and output. The rest of the storage is referred to as *memory*, whether it is on or off-chip. In more recent designs, the on-chip memory that can be accessed by the ALU in a single cycle is referred to as *register file*, with the term memory reserved for off-chip storage. To avoid confusion, we shall use the latter terminology, while referring to the one-bit storage registers of the older machines as *accumulators*.

- **Abacus** [30] – The register file is constructed from dual ported memory and can be read and written during the same cycle. There are two one bit ALUs. Three distinct registers (two input and one output) can be referenced per cycle.
- **ADSP-21020** [35] – The ADSP-21020 is a complete 32-bit DSP chip that can execute 120MFlops per second. It is notable here because it can be configured to run as a PE in a SIMD processor. Because of its complexity, it can be viewed as the upper end of MPA PE complexity.
- **AIS-5000** [125] – Unlike the other processors, the AIS-5000 is a linear array. However, its one bit ALU is representative of the class we are considering. The register file can be read and written during the same cycle. Three registers can be referenced per cycle.
- **CAAPP** [147] — The first generation CAAPP has a very simple datapath, containing a one bit ALU. It has 5 accumulators and can reference a register and an accumulator in the same cycle. There is also an 8 bit internal datapath that speeds up register to register transfers.
- **CM1/2** [140] – The CM1 and CM2 each have a flat memory space, i.e. they have no on-chip storage (register file). Each instruction consists of three memory operations: two reads and a write. The arithmetic operation is executed concurrently with the memory operations by the 1 bit ALU. The CM2 has floating point co-processors.
- **DAP** [4] – Only one register operand can be read or written per cycle, although that operand can be combined with an accumulator during the same cycle. The ALU is one bit wide.
- **GAPP** [44] – Like the CM1 and CM2, accumulators must be loaded and unloaded explicitly to use the ALU and the arithmetic operation takes place concurrently.
- **ILLIAC-IV** [54] – There are 5 64 bit registers and all internal datapaths are 64 bits wide. Features include floating point support (barrel switch, leading one detector, floating point strip circuits) and multiplier support.
- **MP2** [25, 108] – The MP2 PE contains a 32 bit ALU, a 64 bit fraction unit, a 16 bit exponent unit, a 32 bit barrel shifter, a one bit logic unit, and a flag processing unit. The fraction unit doubles as an accumulator. The MP2 allows one register access per cycle.
- **MPP/BLITZEN** [18, 19, 28] – The MPP and Blitzen have a one bit ALU plus a 32 bit shift register. They are also 'accumulator' machines: both operands must be explicitly loaded into the one bit accumulators before the result can be obtained.

The datapath examples summarized above span a wide range of complexity. On the high-end are the ILLIAC-IV and the Analog Devices ADSP-21020. On the low-end are the many processors with one bit ALUs and little additional support.

The datapath widths range from 1 to 64 bits. Some processors have differing datapath widths for ALU operations and data transfers. The number of registers that can be accessed per instruction range from 1 to 3. The features available include shifters, barrel shifters, other explicit floating point support including coprocessors, and multiplication support.

The PE datapath, more than the other components of the MPA design, is the subject of the complexity/number tradeoff: should there be a large number of simple PEs, or should there be fewer but more complex PEs? In particular, has a trend developed in one direction or the other?

Historically, most machines have had a number of processing elements on the order of the size of an image. Because of technological constraints, this has usually meant one bit ALUs. However, even with the continuing advances in VLSI and packaging, this 'minimalist' PE strategy is still being researched in MPAs such as the Abacus and the MGAP [80, 112]. But clearly the technological advances have made complex PEs an attractive option, as MasPar has shown with the MP1 and MP2.

We conclude this discussion by stating that no consensus has yet been reached as to the appropriate granularity of MPAs. Therefore the datapath evaluator must be able to support the entire range of possibilities. We review its design in the next section.

8.2 Datapath Evaluator Overview

Although we have just shown that MPA PE datapaths can contain a wide variety of features, they are still substantially simpler than, say, modern RISC processors. In particular, the datapaths are not pipelined. In large part, the simplicity of MPA PEs results from not having micro-sequencers: there is no need to fetch instructions, do instruction decode, or shift the flow of control.

The consequence of MPA datapath simplicity for this evaluation study, is that the datapath design is not sensitive to instruction order and so does not require a separate trace-driven simulator. Rather, the datapath evaluation is integrated into the code generation phase (Pass 7) of the trace compiler. And since the target machine code is not used for further processing, the output need not even be another trace; instead, the performance information for the application code executing on the target machine datapath is provided immediately.

The datapath design space has been distilled into a parameterized model of a generic MPA, the details of which will be given in the next Section. This parameterized MPA model has a matching instruction set—or more precisely, range of instruction sets—from which the appropriate target machine instructions are generated. Which instructions are generated depends on the target machine parameters.

The flow of the datapath evaluator is shown in Figure 8.1. The user creates a datapath model for the target machine by inputting the target machine features and parameters into the ***virtual machine/target machine dictionary generator***. The dictionary generator uses the generic MPA PE datapath model to generate the particular ***virtual machine/target machine dictionary***, which specifies how to translate virtual machine instructions into target machine code instructions. The

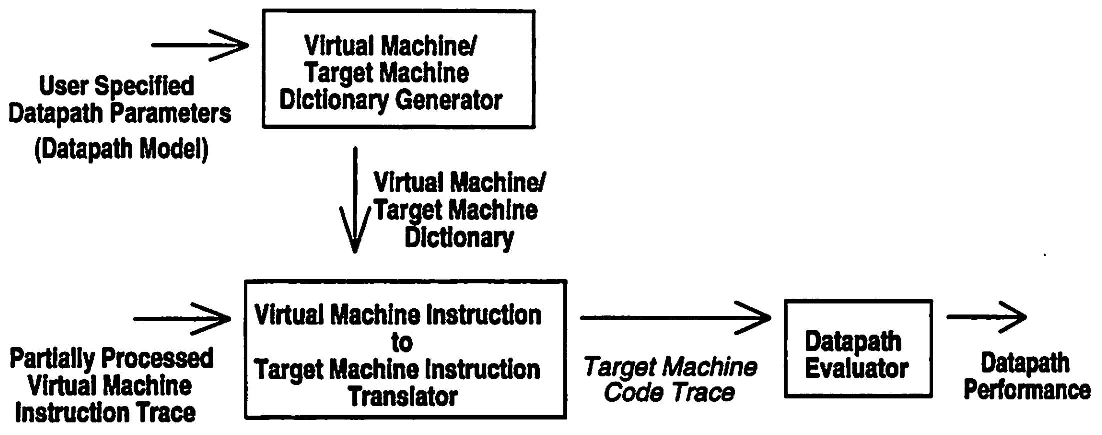


Figure 8.1. Datapath Evaluator block diagram. The input trace is the output from Pass 6 of the trace compiler. The target machine code trace is only required for pipelined datapath evaluation.

virtual machine to target machine translator takes as input the dictionary and the logical machine instruction trace from Pass 6 of the trace compiler. It performs the actual translation, counts the cycles, and outputs the datapath performance results.

In the future, when more complex datapath designs are evaluated, the virtual machine to target machine translator will output a ***target machine code trace***. This trace will then need to be processed further, for example for pipeline evaluation, by the ***datapath evaluator***.

8.3 The Parameterized Datapath Model

Although it would be desirable for the parameterized MPA datapath model to subsume every possible datapath design attribute, this is clearly impractical. We do, however, include in the model most of the attributes available in the existing processors in the design space.

We also make several assumptions to limit the design space and thereby bound the complexity of the model. One is that future MPAs will no longer be 'accumulator' machines, i.e. all future MPAs will be able to read and process at least one register per cycle. We also assume a RISC instruction set, i.e. that most of the instructions in the target processor will be either load/store, arithmetic, or communication. Since we have SIMD control, there are no flow control instructions at the PE level.

Another assumption is that the register file will be uniform. Although there can be special floating point registers, for example, we do not support the menagerie of Q, V, E, A, B, C, Z, X, Y, P, etc. registers prevalent in many of the older designs. They are supported, however, if they can be abstracted into a single accumulator, or if they provide an essential function such as activity control.

One final point before we present the details of the model: it turns out that this last stage of trace compilation is a convenient place to evaluate other MPA components as well. There is also some overlap in what is done in the system, array, and datapath. Therefore the generic datapath model also includes other architectural components such as the nearest neighbor network and array to controller feedback.

We begin by describing the system model, followed by the array model and finally the datapath model.

8.3.1 System Model

The MPA system model has the following characteristics:

- a controller which broadcasts instructions and immediate data to the array,
- an array of PEs, and
- feedback circuitry from array to controller in the form of a global OR of the responders.

The system model includes the following optional features and parameters:

- The number of PEs in the array can vary within the constraint that it be on the same order as common image sizes, and
- feedback circuitry from array to controller in the form of a global count of the responders.

The system model is shown in Figure 8.2.

8.3.2 Array Model

The MPA array model has the following characteristics:

- PEs are connected with a mesh router network.

The array model includes the following optional features:

- floating point units (if there is a floating point unit for every PE, then this feature is described in the ALU model)
- an additional dedicated router network.

The array model includes the following design parameters:

- the path-width and latency of the nearest neighbor network,
- the number of floating point units per PE, if less than one unit per PE,
- the type of corner turning, load/unload hardware (if there is a corner turning unit for every PE, then this feature is described in the ALU model), and
- the type of communication network (model described elsewhere).

The system model is shown in Figure 8.3.

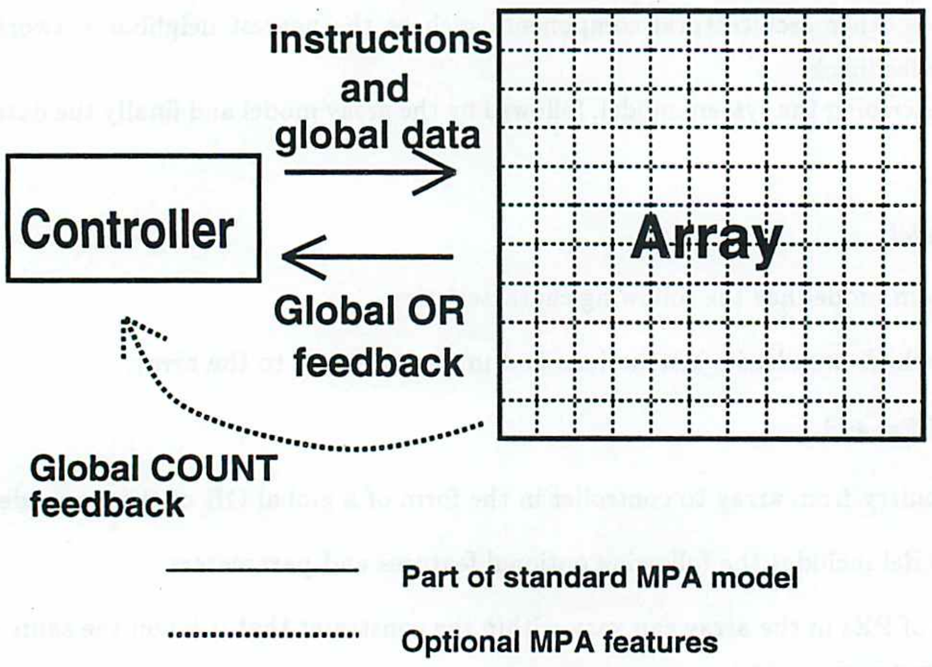


Figure 8.2. The MPA system space.

8.3.3 PE Datapath/ALU model

The generic MPA datapath model is shown in Figure 8.4. The MPA PE datapath model has the following characteristics:

- The PEs all simultaneously execute the identical instruction broadcast by the controller.
- The PEs are simple, non-pipelined units and can only operate on registers, the accumulator, and data broadcast by the controller (which we refer to as *immediate* operands). Machine instructions take a single cycle unless otherwise specified.
- The ALU has two inputs and one output. The inputs can be optionally inverted. The ALU supports ADD, AND, OR, and XOR. If the ALU is wider than 1, then RSHIFT and LSHIFT are also supported.
- Each PE has an accumulator which can be the ALU source, destination, or both in a single cycle.
- Each PE has some number of registers, at least one of which can be accessed by the ALU in a single cycle.
- Immediate operands are also available directly to the ALU. An immediate operand can be either a constant or a scalar variable.

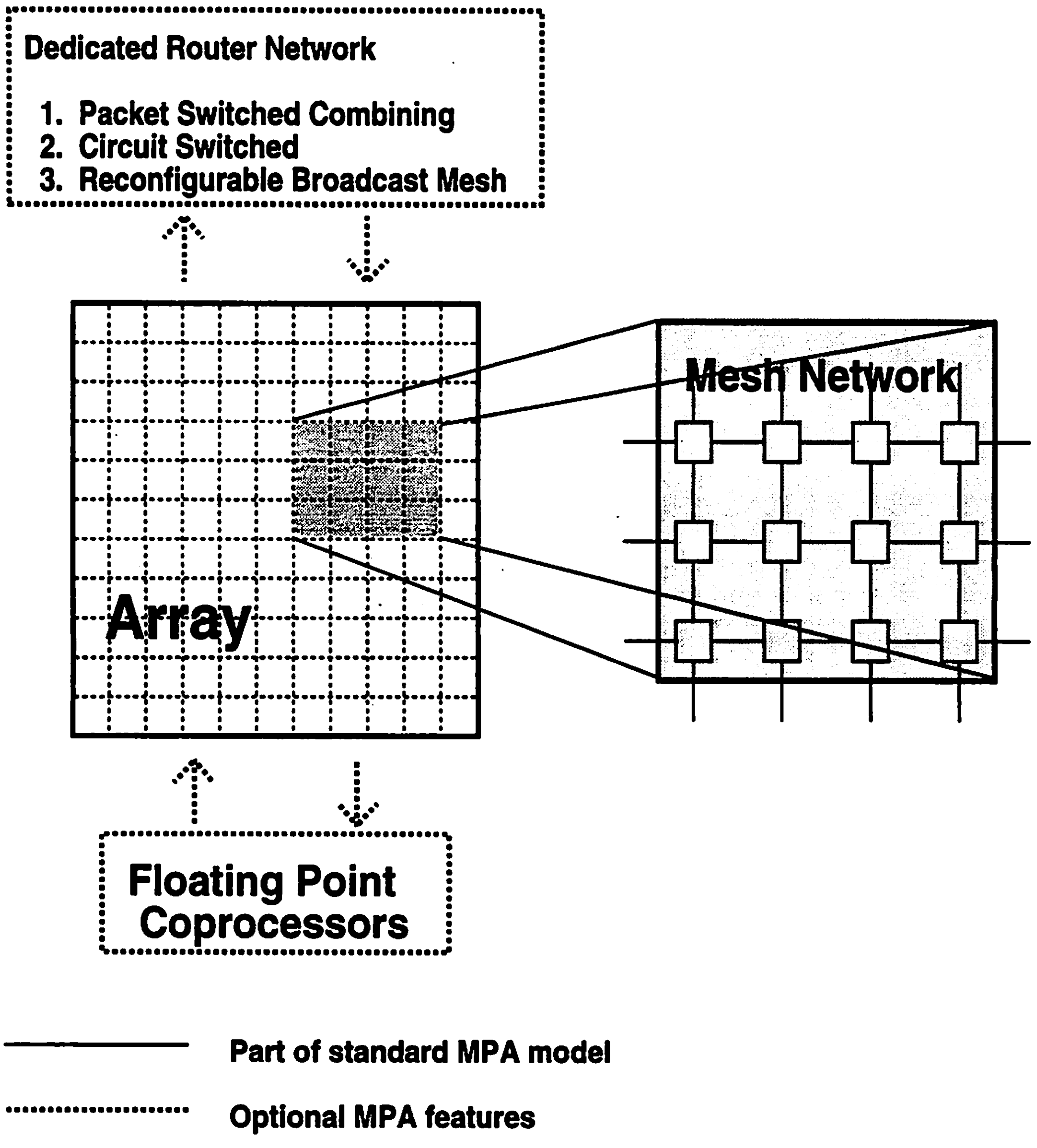
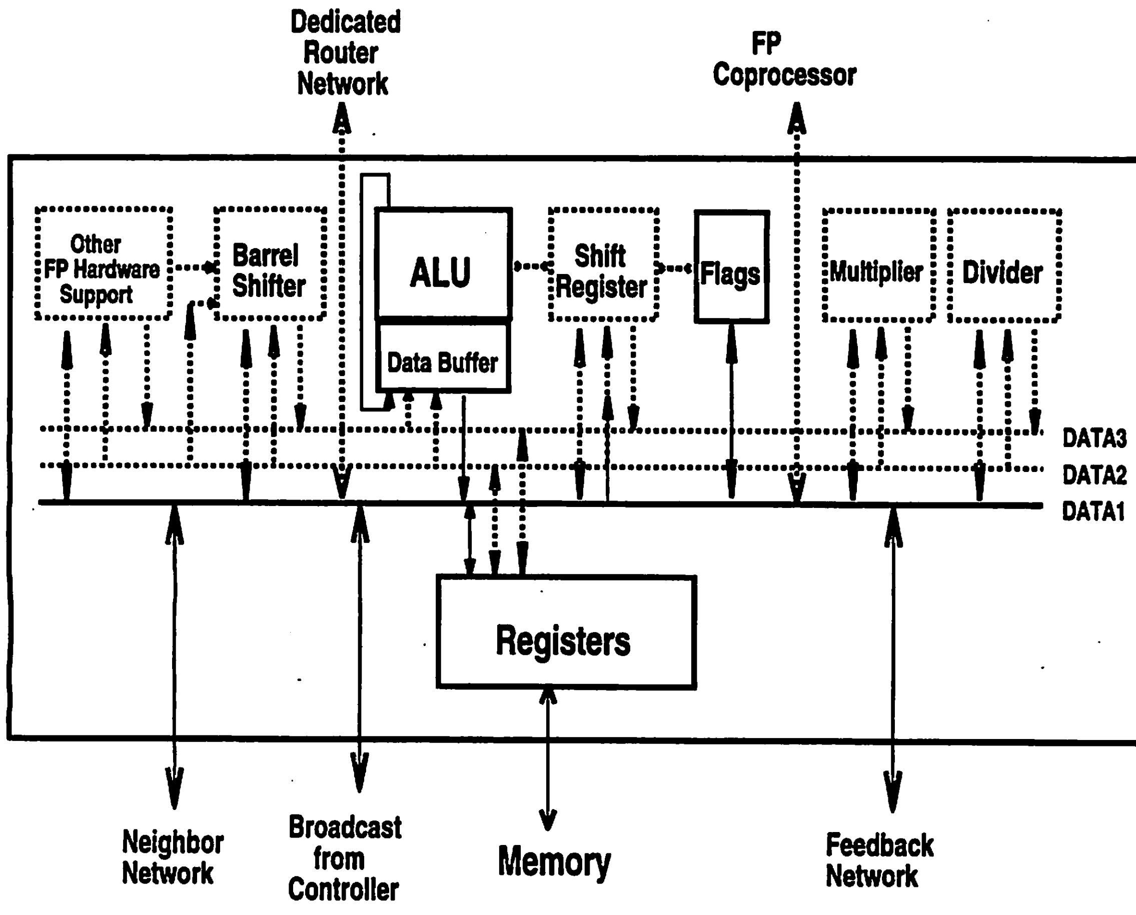

Dedicated Router Network
1. Packet Switched Combining
2. Circuit Switched
3. Reconfigurable Broadcast Mesh
Array
Mesh Network
Floating Point Coprocessors
Part of standard MPA model
Optional MPA features

Figure 8.3. The MPA array space.



Datapath Parameter Space

Number of operands/cycle
ALU width/latency
Datapath width/latency
Shift Register?
Multiplier?
Divider?
Barrel Shifter?
Floating Point Support?

Defining Characteristics	———
Optional Features	•••••••••

Figure 8.4. The generic MPA ALU/datapath model.

1.	OP	(¬)RegA,(¬)Acc,RegA
2.	OP	(¬)RegA,(¬)Acc,Acc
3.	OP	(¬)Imm,(¬)RegA,RegA
4.	OP	(¬)Imm,(¬)RegA,Acc
5.	OP	(¬)Imm,(¬)Acc,RegA
6.	OP	(¬)Imm,(¬)Acc,Acc
7.	OP	(¬)RegA,(¬)RegB,Acc
8.	OP	(¬)RegA,(¬)Acc,RegB
9.	OP	(¬)RegA,(¬)RegB,RegA
10.	OP	(¬)RegA,(¬)Imm,RegB
11.	OP	(¬)RegA,(¬)RegB,RegC

Table 8.1. The legal ALU instruction templates.

- There is enough spare register file to be used as scratch space for floating point operations, the local copy of the PE ID, and array or tile edge status.
- Registers wider than the ALU are addressable in units the size of the ALU width.
- If the ALU is operating on data whose type-size is smaller than that of the ALU, sign extension is supported as needed.
- The datapath/ALU model comes in one of three flavors, depending on the number of different registers that can be accessed during each cycle. The instruction templates can be found in Table 8.1. 1-6 are for 1 address machines, 1-10 are for 2 address machines, and 1-11 are for 3 address machines.
- If the ALU is wider than 1, then each PE also has a processor status word with the following status bits: C,N,Z,V. These are set, according to standard convention, on carry (C), overflow (V), on a zero result (Z) and on a negative result (N). Further, hardware support exists to write combinations of these status bits into the accumulator in a single cycle, e.g. to implement conditional operations.
- On shift operations, the quantity shifted into the operand can be either the carry bit, a zero, or the sign. The quantity shifted out can optionally be put into the carry bit.
- PEs have links to the router network, floating point units and feedback circuitry. The types of links depend on the particulars of those components.

The datapath model includes the following optional features:

- # of register operands per cycle
- Width of ALU (possible sizes: 1,2,4,8,16,32,64)

- Width of datapath (possible sizes: 1,2,4,8,16,32,64 and wider than the ALU). The datapath can be wider than the ALU. In this case, there must exist a datapath accumulator or Dacc. It does not make sense for the datapath to be narrower than the ALU in this model since memory references cannot be used as operands.

- ALU support of BitTypes. This is a subtle point: BitTypes are stored in byte quantities. Mixed operations between BitTypes and other types, are simplified if the high order seven bits can be ignored (e.g. they do not affect the sign).

- Multiple add instructions: one that does and the other that does not add the carry bit to the two input operands. This saves a cycle in situations where the carry would otherwise need to be cleared or set before the arithmetic function.

- Shift register independent from the ALU (possible sizes: 16,32,64,128). This is useful for multiplication and division.

- Hardware multiplier (possible sizes: 4,8,16,32,64).

- Hardware divider (possible sizes: 16,32,64).

The datapath model also includes the following optional features that support floating point operations:

- Barrel shifter independent of ALU and shift register. The barrel shifter takes two operands as input, the value to be shifted, and the amount. Latency in cycles is a parameter.

- Leading one detector. One input and two outputs. Shifts operand to the left until a one appears in the high order bit and records how many shifts were needed. Latency in cycles is a parameter.

- Floating point registers. Hardware is assumed to do single cycle alignment into and out of the temps in which the arithmetic operations are executed.

- Parallel sign operations. The signs are calculated in parallel with the rest of the operation.

- Floating point co-processor. Parameters are the latencies of each operation, and the time needed to do corner turning, load, and unload. These parameters are specified in the array model if there is less than one FP co-processor per PE.

- Double floating point co-processor. Same as above.

8.4 Datapath Evaluator Details

We now present the details of the datapath evaluator components.

8.4.1 The Input Trace

The input to the datapath evaluator is the output of Pass 6 of the trace compiler (see Section 7.7.6). Recall the state of the trace at that point. The trace, comprised of MVM instructions, was generated by virtual machine emulation. It was then passed through a series of transformations. Some optimization has taken place, especially to minimize the number of temps used. Physical memory and register addresses have been replaced by logical machine variable names. The Allocate and Deallocate directives inserted by the virtual machine emulator have been removed and Load and Store instructions have been added. However, the bulk of the instructions are still in the form found in the MVM instructions (see Tables 7.1 and 7.2 on Pages 75 and 76).

8.4.2 The Input Model

The input model for a particular target machine is the set of parameters and features that the user inputs to the virtual machine/target machine dictionary generator. The allowable features and the ranges of parameters were described in the previous section. A sample model for a CAAPP-like datapath is in Table 8.6 on Page 113.

8.4.3 Dictionary Generator

Creating a particular datapath dictionary d means creating a translation for every MVM instruction into d-machine code instructions. The function of the datapath dictionary *generator* is to create the MVM to d_P-machine dictionary for any legal set of parameters P. The idea is to produce routines in the target's machine language to execute MVM instructions on the hardware specified in the model. The dictionary generator is thus a code generator generator.

Some of the relationships between design feature (or parameter) and virtual machine instruction instantiation are simple. For example, the MVM ADD instruction complexity is directly related to the ALU width: if the ALU width is smaller than that of the operands, then the operands are simply fed through the ALU in ALU-width chunks from low-order to high-order. Other relationships between datapath model and MVM translation are more complex, for example, the effect of the presence or absence of a shift register on floating point instructions requires generating and counting the target machine instructions.

The major problem with producing the virtual machine/target machine dictionary generator is the large number of variations that each virtual machine instruction can take. For the integer instructions, each operand can be any legal width (1,4,8,16,32,64) and either signed or unsigned. Also, one of the input operands can be a scalar (global data) broadcast by the controller.[1]

For the rest of this subsection, we show some of the hardware dependencies for various virtual machine instructions. We also show in detail how the dictionary entries are generated for two instructions—ADD and SHIFT—for various target hardware configurations. Even here,

[1] If both of the operands are scalars, then the entire operation would be performed in the controller.

the cases where the operands have different types will not be presented. Notice that the more complex virtual machine instruction instantiations use simpler ones as 'subroutines.' To make this distinction, square brackets will be used to denote virtual machine instructions and parentheses to denote target machine instructions.

8.4.3.1 Arithmetic, Logical, and Set Instructions

The PE designs for MPAs tend to be very simple with just enough complexity to implement a RISC-like instruction set. The ALU is assumed to be able to process the standard arithmetic, logical, and register-accumulator transfer operations in a single cycle. The hardware dependencies are

- number of registers referenced per cycle (1,2, or 3),
- width of ALU (1,2,4,8,16,32, or 64), and
- whether it is possible to set and clear the condition codes in parallel to an instruction execution.

Below is pseudo-code for the hardware dependent [ADD, A, B, C] virtual machine instruction where both operands have the same integer type. The other instructions are analogous.

```
[ADD, A, B, C]
IF the carry flag cannot be cleared in parallel with the first [ADD, A, B, C]
      instruction, then it is cleared explicitly with the (CLR C-flag) operation.

- - one of the following loop bodies is executed

DO Width[A]/Width[ALU] TIMES
      Case (One register reference per cycle, both operands are planes):
            (MOV, Ai, ACC)
            (ADD, Bi, ACC, ACC)
            (MOV, ACC, Ci)
      Case (One register reference per cycle, one operand is a plane,
      the other a scalar):
            (ADD, Ai, Imm, ACC)
            (MOV, ACC, Ci)
      Case (Two register references per cycle, both operands are planes):
            (ADD, Ai, Bi, ACC)
            (MOV, ACC, Ci)
      Case (Two register references per cycle, one operand is a plane,
      the other a scalar):
            (ADD, Ai, Imm, Ci)
      Case (Three register references per cycle, both operands are planes):
            (ADD, Ai, Bi, Ci)
      Case (Three register references per cycle, one operand is a plane,
```

the other a scalar):

```
    (ADD, Ai, Imm, Ci)
```

Since all of the above operations are single cycle, the resulting number of cycles for the ADD instruction as a function of the target machine parameters is simply determined.

8.4.3.2 Left and Right Shift

If there is a barrel shifter that is wide enough to hold A, then it is loaded with A and B, the shift is executed, and the barrel shifter is unloaded into C. If the operand A has more bits than the ALU or the shift register, then the individual single-bit shift operations are instantiated with shifts of part of the operand with the C-flag being used as the input and output of each operation.

The [LEFTSHIFT, A, B, C] and [RIGHTSHIFT, A, B, C] instructions shift the variable A by the amount in variable B and leave the result in C. For a SIMD array with no barrel shifter, shift instructions can be a very costly since any PE can possibly shift any number of places, up to the number of bits in A. Therefore, if there is no barrel shifter, a counter must be used to determine that the A operand has shifted the correct distance in each PE.

The SHIFT instructions depend on the same architectural parameters as the arithmetic, logical, or set operations. They also depend on the presence (or absence), size, and latency of a shift register or barrel shifter.

If the number of bits to be shifted is a scalar, then the operation is simplified considerably. In particular, the total number of single bit shifts should never exceed the width of the shift register minus one. The rest of the shift can be performed using MOVs.

```
[SHIFT, A, B, C]
IF BarrelShifterWidth > Width(A)
    DO Width(A)/Width(DataPath) TIMES
        IF One register reference per cycle
            (MOV, Ai, BarrelShifterData)
            (MOV, Bi, BarrelShifterDistance)
        IF Two register references per cycle
            (MOV, Ai, Bi, BarrelShifter)
    (SHIFT, BarrelShifter)
    DO Width(A)/Width(DataPath) TIMES
        (MOV, BarrelShifterData, Ci)
ELSE IF A is a variable
    IF ShifterWidth >= Width(A) && Width(ALU) >= Width(B)
        DO Width(A)/Width(DataPath) TIMES
            (MOV, Ai, Shifter)
            (MOV, Bi, Accumulator)
        DO Width(A) TIMES
            (DEC, Accumulator)
            (MOV, Z-flag, A-flag)
```

```
                    (SHIFT, Shifter)
                DO Width(A)/Width(DataPath) TIMES
                    (MOV, Shifter, Ci)
            ELSE IF ShifterWidth >= Width(A) && Width(ALU) < Width(B)
                [SET, B, TEMP]
                DO Width(A)/Width(DataPath) TIMES
                    (MOV, Ai, Shifter)
                DO Width(A) TIMES
                    [DEC, TEMP]
                    (MOV, Z-flag, A-flag)
                    (SHIFT, Shifter)
                DO Width(A)/Width(DataPath) TIMES
                    (MOV, Shifter, Ci)
            ELSE IF No Shifter && Width(ALU) >= Width(B)
                [SET, A, C]
                DO Width(A)/Width(DataPath) TIMES
                    (MOV, Bi, Accumulator)
                DO Width(A) TIMES
                    (DEC, Accumulator)
                    (MOV, Z-flag, A-flag)
                    [SHIFT, C, 1]
            ELSE                            – No Shifter && Width(ALU) < Width(B)
                [SET, A, C]
                [SET, B, TEMP]
                DO Width(A) TIMES
                    [DEC, TEMP]
                    (MOV, Z-flag, A-flag)
                    [SHIFT, C, 1]
        ELSE                                – B is a scalar
            IF ShifterWidth >= Width(A)
                DO Width(A)/Width(DataPath) TIMES
                    (MOV, Ai, Shifter)
                DO Imm TIMES
                    (SHIFT, Shifter)
                DO Width(A)/Width(DataPath) TIMES
                    (MOV, Shifter, Ci)
            ELSE IF ShifterWidth >= Width(A) && Imm < Width(ALU) - 1
                [SET, A, C]
                DO Imm TIMES
                    [SHIFT, C, 1)
            ELSE IF ShifterWidth >= Width(A) && Imm < Width(ALU) - 1
                DO Imm/Width(ALU) TIMES
                    [SET, A, C]
                DO Imm%Width(ALU) TIMES
                    [SHIFT, C, 1]
```

All of the above operations are single cycle, except the barrel shift which requires a latency as specified in the target machine model. Also note the data dependency when the shift amount is a scalar and the ALU is small.

8.4.3.3 Bit Operations

There are two virtual machine instructions that operate on bits within a variable, BIT and INSERTBITS. BIT takes three arguments, the input PLANE, the output BITPLANE, and a scalar that denotes which bit is to be extracted from the input and moved to the output. INSERTBITS directs the movement of a contiguous subset of bits from one PLANE to another. The instruction takes five arguments, the input PLANE, the output PLANE, and three scalars. The scalars denote the starting point of the bit sequence in the input, in the output, and the length of the bit sequence. The translation of the INSERTBITS instruction will be discussed here, the BIT instruction will not be discussed here because it follows directly from INSERTBITS.

The following is a generic INSERTBITS procedure:

```
INSERTBITS(A,B,A_start,B_start,length)
1. [SET, A, Temp]
2. [ANDEQ, Temp, Imm]            - Strip desired bits
3. [SHIFT, Temp, Imm, Temp]      - align bits
4. [ANDEQ, B, Imm]               - clear output slot
5. [OREQ, B, Temp]               - insert bits
```

The bit instructions clearly have the same hardware dependencies as the SHIFT instructions.

8.4.3.4 Multiplication

Since we are describing the instantiation of an instruction applied simultaneously to a large number of PEs under SIMD control, data dependent techniques such as Booth's algorithm do not result in any performance gain. Therefore, unless there is a dedicated multiplier in the specification, the standard add-shift technique is used (see for example [41]).

The multiply instruction depends on similar hardware as the shift operations, with the following exceptions: since only single bit shifts are executed, there is no benefit to having a barrel shifter; and having a combinational multiplier of even small size increases performance substantially by generating the partial products.

The effect of varying the ALU/Datapath widths and of adding a 32-bit shift register are shown in Table 8.2 and Figure 8.5. Note that in the absence of a shift register, going from a 1 bit to a 2 bit ALU can actually be a detriment. This is because bit-serial processors have the capability of executing a shift operation implicitly by changing the operand address. Note also that without additional hardware support, the multiply still takes 32 times a constant number of cycles.

The effect of adding a dedicated multiplier circuit is shown in Table 8.3 and Figure 8.6. Note particularly that substantial benefit is obtained even when less than a full multiplier is used. For

Architectural Features	ALU/Datapath Width					
	1	2	4	8	16	32
1 reg. op/cycle	2507	2200	1120	580	310	280
2 reg. ops/cycle	1295	1608	824	432	236	138
1 reg. op/cycle, 32 bit shifter	2507	1720	896	484	278	114
2 reg. ops/cycle, 32 bit shifter	1295	1144	608	340	206	107

Table 8.2. Execution times of the 32-bit multiply instruction as a function of ALU/datapath width as the number of register operands per cycle is varied and a 32 bit shifter register is added.

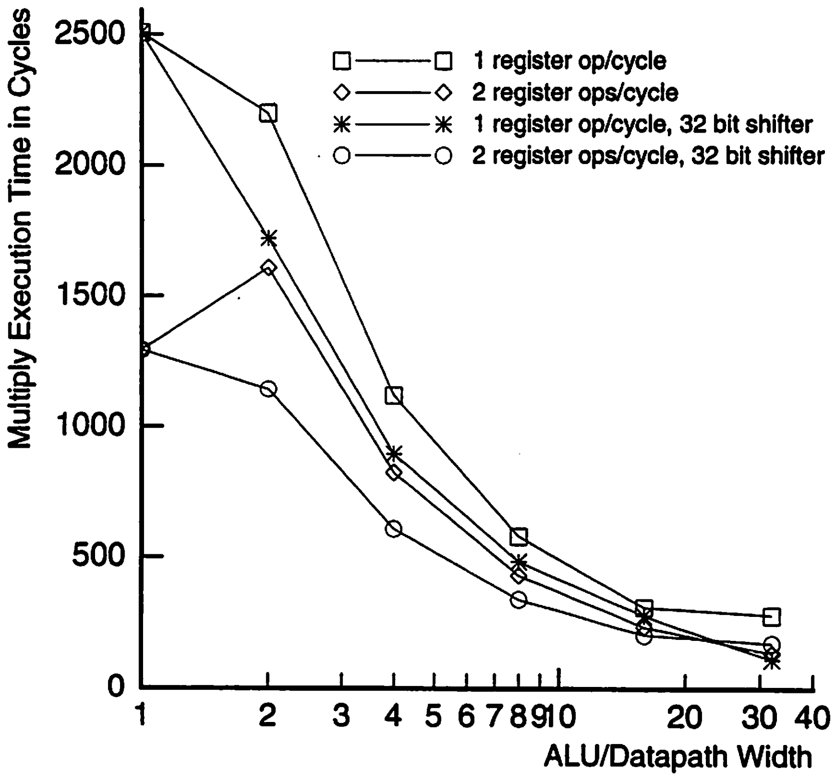


Figure 8.5. The effect of varying the ALU and internal datapath widths, and presence of a 32-bit shift register on the absolute execution time of the 32-bit integer multiply.

Architectural Features	Multiplier Size				
	none	4	8	16	32
1 reg. op/cycle, 1 bit ALU/Datapath	2507	464	272	176	128
1 reg. op/cycle, 4 bit ALU/Datapath	912	120	68	42	28
2 reg. ops/cycle, 4 bit ALU/Datapath	824	108	60	36	24
2 reg. ops/cycle, 32 bit ALU/Datapath	107	72	20	6	2

Table 8.3. Execution times of the 32-bit multiply instruction as a function of multiplier size and ALU/datapath width.

example, a 4 bit circuit speeds up the 4 bit ALU/Datapath multiply by a factor of 8, so that it is less than a factor of 8 slower than a 32 bit integer add. Similarly, the 8 bit and 16 bit multipliers speed up the 32 bit ALU/Datapath multiply by factors of 5 and 18 respectively. Since multipliers require chip area that is quadratic in the number of bits, the fact that comparatively small multipliers (having smaller width than the ALU/Datapath) still yield substantial performance improvements is likely to be significant.

8.4.3.5 Division

Just as with multiplication, the SIMD control makes the division operation more time consuming than it would be on a serial processor: standard techniques that avoid the restore step do not work.

- Restoring division using multiplexing is no faster than standard restoring division because the register-register copy is no faster than an ADDEQ (+ =) operation where the accumulator is both one of the inputs and the output.

- Non-restoring division is also no faster than restoring division because, in the crucial step, there is a data dependent choice between adding or subtracting the divisor from the dividend. Because of the SIMD regimen, both of these choices must be executed during every iteration.

- Other data dependent techniques, such as SRT division, also do not improve performance for the same reason.

Division depends on the same hardware as the multiply instruction, with the exception that a dedicated divider, rather than multiplier, is needed. In the latter case, however, an analogous benefit to that of a partial multiplier is not accrued. This is because of the inherent non-determinism in division.

Since the effect of varying architectural features for division is very similar to that for multiply, detailed results are not presented.

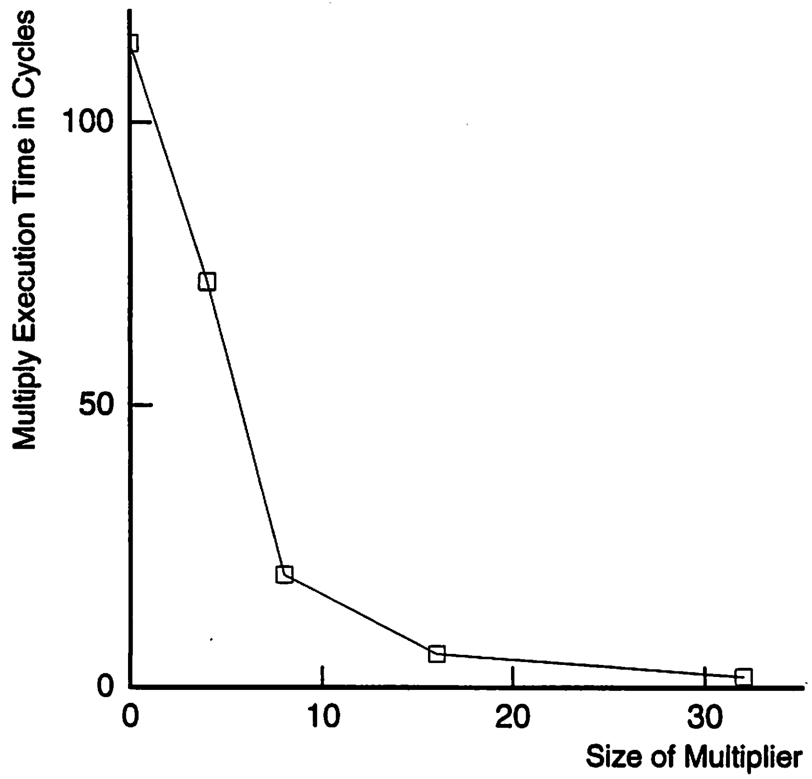


Figure 8.6. Effect of adding dedicated multiplier circuit of various sizes on the execution time of the 32-bit integer multiply. Zero on the X-axis indicates no multiplier.

8.4.3.6 Floating Point Operations

32-bit and 64-bit floating point types are supported. There are two basic scenarios: the target machine has floating point co-processors or it does not. If floating point co-processors are available, then the dictionary entries for the floating point operations are simply the latencies of the operations that were specified by the user. However, if there is no complete and transparent floating point support, then the floating point operations depend on the same hardware as what is available for processing the integer operations, plus certain other components. These components include barrel shifters, leading one detectors, special registers for floating point operands, and the capability of processing several parts of the instruction (e.g. the sign) in parallel.

The data in Table 8.4 show the effects of various architectural parameters and features on floating point addition. Since much of the complexity of computing floating point instructions on integer hardware comes from stripping bits and shifting data, adding a barrel shifter results in a substantial performance gain. Because much of the remaining complexity comes from the need to normalize the result, the performance gain is magnified when a leading one detector is also added.

An important observation about the data in Table 8.4 is that even with the hardware support just mentioned, floating point addition is still a costly instruction. This is largely a simple

Architectural Features	ALU/Datapath Width					
	1	2	4	8	16	32
2 reg. ops/cycle	2318	1775	919	503	372	376
2 reg. ops/cycle, 32 bit barrel shifter	1061	731	429	291	244	179
2 reg. ops/cycle, 32 bit barrel shifter, leading one detector						85

Table 8.4. Execution times of the single precision floating point add instruction as a function of ALU/datapath width and various other features.

Architectural Features	ALU/Datapath Width					
	1	2	4	8	16	32
2 reg. ops/cycle	1712	1596	838	471	292	196
2 reg. ops/cycle, 32 bit barrel shifter	1712	1238	672	401	270	139
2 reg. ops/cycle, 32 bit barrel shifter, 4 bit multiplier	944	638	348	233	198	138
2 reg. ops/cycle, 32 bit barrel shifter, 8 bit multiplier						87
2 reg. ops/cycle, 32 bit barrel shifter, 32 bit multiplier						69

Table 8.5. Execution times of the single precision floating point multiply instruction as a function of ALU/datapath width and various other features.

matter of there being many functions that need to be performed. What is necessary to improve performance further is to have the ability to perform some of these functions in parallel. For example, a separate sign processor is helpful.

The results in Table 8.5 show the effects of various architecture parameters and features on floating point multiplication. Since the result of this instruction is simpler to normalize than that of the floating point add, the leading one detector is not an advantage. However, an integer multiply unit naturally provides a substantial speed up. The floating point division instruction is analogous.

8.4.4 Virtual Machine/Target Machine Dictionaries

For the relatively simple PEs we are considering in this study, the virtual machine/target machine dictionaries are primarily look-up tables containing the latencies of the legal MVM-instruction/operand-type combinations. The exception is the entries for the MVM instructions whose execution times can be data dependent. In this category are the routing instructions (which will be discussed in detail in Chapter 10) and the INSERTBITS instruction. The dictionary entries for the data dependent instructions contain sufficient information to be used by the translator to compute the execution time as the corresponding trace record is processed.

For an example, see Section 8.5 where a CAAPP-like processor is defined in Table 8.6 and where part of the resulting virtual machine/target machine dictionary is shown in Table 8.7.

8.4.5 Translator (Evaluator)

The virtual machine instruction to target machine instruction translator takes as input the MVM/target machine dictionary and the logical machine trace (Pass 6 output) and outputs datapath performance data. Data independent instructions are evaluated immediately through table lookup. Data dependent instructions are processed with respect to the architectural parameters—both those passed in with the dictionary and with the trace—as well as the data included in the instruction record.

For example, if the hardware includes a shift register but not a barrel shifter, then the execution time of the [SHIFT, A, Imm] instruction (i.e. shift by a constant) depends on the value of Imm, or the number of slots the variable must be shifted. To handle this data dependency, the value of Imm is explicitly recorded at emulation time and saved along with the MVM instruction record. There it remains as the trace is processed.

8.5 Evaluating Existing Machines: An Example

Recall that there are two primary uses of ENPASSANT:

1. to be a tool in examining the basic issues in MPA design such as processor granularity and the complexity of the inter-PE communication network, and

2. to study existing designs, that is, benchmarking and examining the effects of future parameter and feature changes.

These two uses obviously overlap: for example, existing designs are naturally good starting points for searching the MPA architecture space.

For a general evaluation system to have the capability of completely and precisely modeling more than a single existing design, however, is problematic and perhaps not even desirable. Modeling even one machine precisely is a very complex task, and often impossible if it is not done in conjunction with the machine's designers. There are invariably 'unadvertised features,' that are either trade secrets or too obscure to appear in the literature, yet which affect the performance. Including machine specific features from multiple target machines—even if information about them is publicly available—in a general evaluation system is likely to lead at the very least to unmanageable complexity, and at most to misleading conclusions. The latter can occur when features that were the result of particular circumstances are included in a potential design where those circumstances are unlikely to exist.

Register Operands per Cycle	1
ALU Width	1
Datapath Width	8
ALU Supports Bit Operations?	Yes
ALU Supports ADD and ADDC Ops?	Yes
Floating Point Hardware?	No
OR Feedback Latency	3
COUNT Feedback Latency	20
Nearest Neighbor Set Up Time	0
Nearest Neighbor Path Width	1
Nearest Neighbor Transfer Latency	1
Coterie Network Set Up Time	1
Coterie Network Path Width	1
Coterie Network Transfer Latency	10
Load/Store Latency	5

Table 8.6. Parameters that comprise the CAAPP-like model

Rather we have taken a slightly different approach. In the ENPASSANT datapath model we have abstracted the key features from the class of MPA designs; features whose availability is relatively easy to determine. This set of features is certainly sufficient to model most of the relatively simple PE designs found in MPAs. If the ENPASSANT datapath model is not sufficient to model a particular machine, an over-ride mechanism is provided. To use this mechanism, the user first models the target machine as accurately as possible. Then, wherever an MVM instruction has not been adequately modeled, the user inputs a more accurate value.

We now present as an example a CAAPP-like model (see Table 8.6) and the dictionary that is built from it (see Table 8.7). The dictionary entries are compared with data derived from the detailed CAAPP simulator.

The differences are mostly the product of two factors, one having to do with an architectural ideosyncracy of the CAAPP, the other with the choice of arithmetic algorithm.

- The CAAPP uses the Coterie Network switch register during data movement. This register must be restored to its original value at the end of every instruction where it is used, adding one to the cycle count.

- The CAAPP microcode for the multiply and divide instructions uses arithmetic shifts. However, the time required for the sign extend is expensive in comparison to the simpler method of operating only on positive values and dealing with the sign explicitly. The latter is used by ENPASSANT. These methods are discussed in the appendix of [69].

Virtual Machine Instruction	Execution Time On CAAPP-like Model	Execution Time On CAAPP Simulator
8-bit register-register transfer	2	3
16-bit register-register transfer	4	5
32-bit register-register transfer	8	9
8-bit integer add	19	20
16-bit integer add	37	38
32-bit integer add	73	74
8-bit integer multiplication	195	259
16-bit integer multiplication	581	817
32-bit integer multiplication	2175	2883
8-bit integer divide	304	370
16-bit integer divide	1109	1227
32-bit integer divide	4255	4466
floating point add	1480	1460
floating point multiply	1589	1811
floating point divide	2886	3220

Table 8.7. Some of the instruction times derived from the CAAPP-like model. For comparison, instruction times for the actual CAAPP are shown.

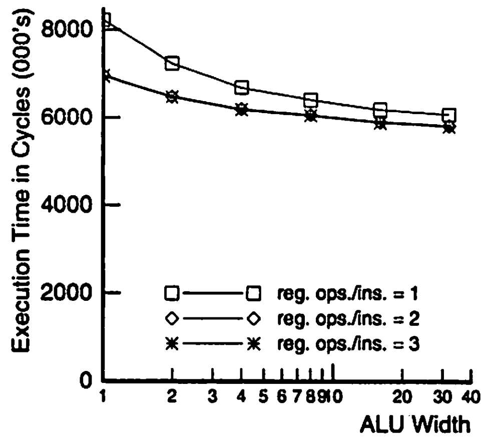
8.6 A Case Study: Varying ALU Width

We now present a case study wherein ENPASSANT is used to determine the effect of varying the ALU and datapath widths on the execution times of several of the test suite programs. Also varied is the number of register operands allowed per target machine instruction. The base model is the CAAPP-like model shown in Table 8.6. Figure 8.7 shows the results.

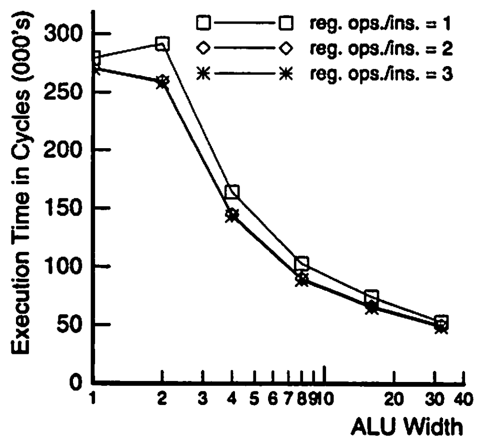
Note that most of the performance gain occurs when the ALU width is increased from 1 to 8, and that, especially, very little performance is gained when the ALU is increased from 16 to 32. This is because most of the operations are on BitPlane, CharPlane, and ShortIntPlane data types. One reason for this data type distribution is that some of the test suite programs frequently operate on 8-bit pixel data. Another reason is that, as in all codes, many operations are inherently binary.

Also note that little gain comes from increasing the number of register operands per instruction from 2 to 3. One reason for this is that relatively few 3 operand instructions remain after the temp-elimination code optimization phase.

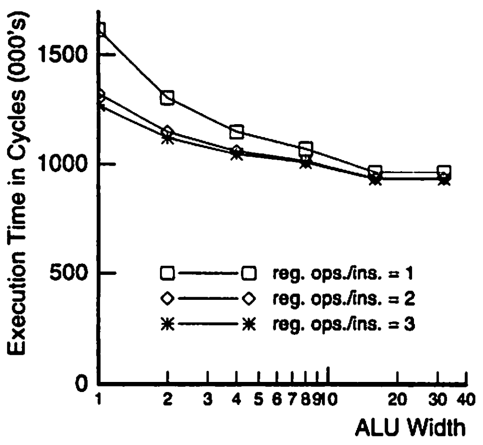
Perhaps the most significant result shown in Figure 8.7 is that for many of the programs, surprisingly little performance improvement results from increasing the ALU and the datapath widths. The exceptions are the Curve-Fitting Filter and Depth From Motion which are dominated by floating computations. For the other programs, there are many virtual machine instructions



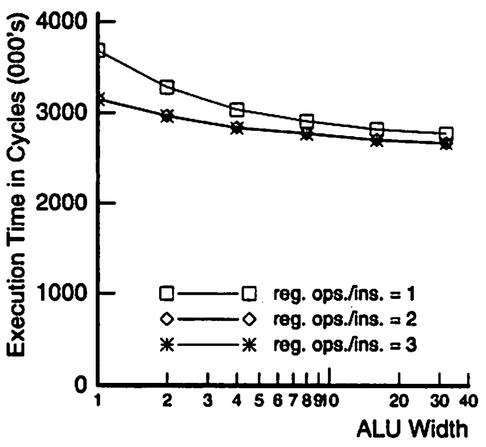
Region-Based Line Finder



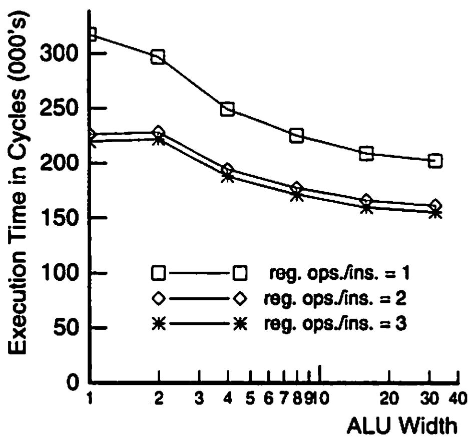
Curve-Fitting Filter



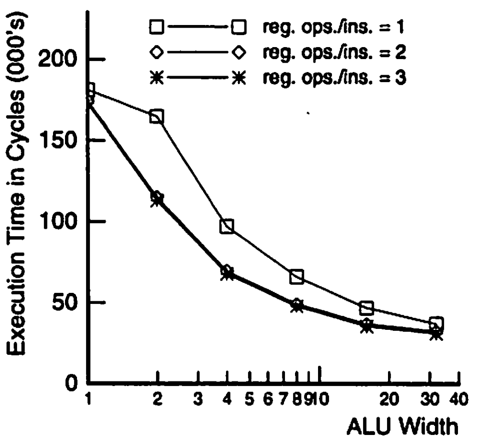
Correspondence Problem



Fast Line Finder



IU Benchmark II



Depth From Motion

Figure 8.7. Effect of varying the datapath width and the number of register operands per target machine instruction. All programs were run on a CAAPP-like model with a register file size of 40 bytes and a virtualization factor of 1.

whose executions times are not decreased significantly by *any* changes in the ALU. In particular, memory references and communication remain costly. These factors will be discussed in the next two chapters.

8.7 Chapter Summary

In this chapter we have presented the datapath analyzer component of ENPASSANT and some examples of how it is used. The datapath analyzer is based on an abstract model derived from current MPAs and their likely extensions. It is thus very flexible and allows for a more detailed examination of the performance implications of variations in datapath designs than was previously available. As examples we have shown some of the architectural dependencies of the multiply, divide, and floating point instructions. We have also shown the relationship between ALU size and the execution time of several of our test suite programs. Somewhat surprisingly, increasing the ALU size from 1 to 32 for a CAAPP-like model only improved performance between 10% and 30% for many of those programs.

Chapter 9

Memory Evaluation

The evaluation of the memory hierarchy can be divided into two components: the register architecture, and the cache/memory architecture. The major difference is that registers are explicitly managed, either statically by the compiler or dynamically by the controller, while caches are managed transparently with supporting hardware. The performance of both components is obviously affected by their access cycle times. However, there are also significant differences which cause different metrics to be useful in measuring their performance, and different methods to be necessary to obtain those measurements.

The critical metric in evaluating the register architecture is the number of load and store instructions required to execute a given program. This quantity depends on the number and type of registers, and also on the quality of the register assignment. For reasons that were mentioned earlier, a certain amount of processing is involved in reconstructing compiler register assignment. If register assignment is done by the controller, then standard cache analysis techniques can be used and the problem is much simpler.

The critical metric in evaluating the cache performance is the average time required per memory reference instruction, which is highly dependent on the cache hit rate. This quantity depends on the size of the cache, the block size, the level of associativity, and, indirectly, on the register architecture. The cache architecture can be analyzed using the standard techniques developed for use with trace-driven simulation.

In the rest of this chapter, we present the MPA memory design space and how we model it, the methods used to evaluate the PE register file and cache architectures, and some sample results.

9.1 The MPA Memory Design Space

One characteristic all the MPA PE memory configurations have in common is that, because of the SIMD control, all array instructions operate on the same register within each PE at the same time. See Figure 9.1 for a common configuration. The same is also largely true for memory locations within each PE; the case of independent local indexing is omitted from this study.

We begin by describing the memory architectures of some typical and well-known MPAs. We again restrict our consideration to machines that have been built relatively recently. Not included in the memory space are the specific accumulator registers in some processors. Recall that we refer to these collectively as the accumulator. The memory sizes indicated are per PE.

- **Abacus** [30] – The Abacus has 4 bytes of on-chip and 2K bytes of off-chip storage. The on-chip storage is loaded at 40 cycles/bit by reading in data from the edge of each chip. Achieving this rate with a 120 Mhz clock is possible because the off-chip memory is high-speed SRAM with a caching mechanism.

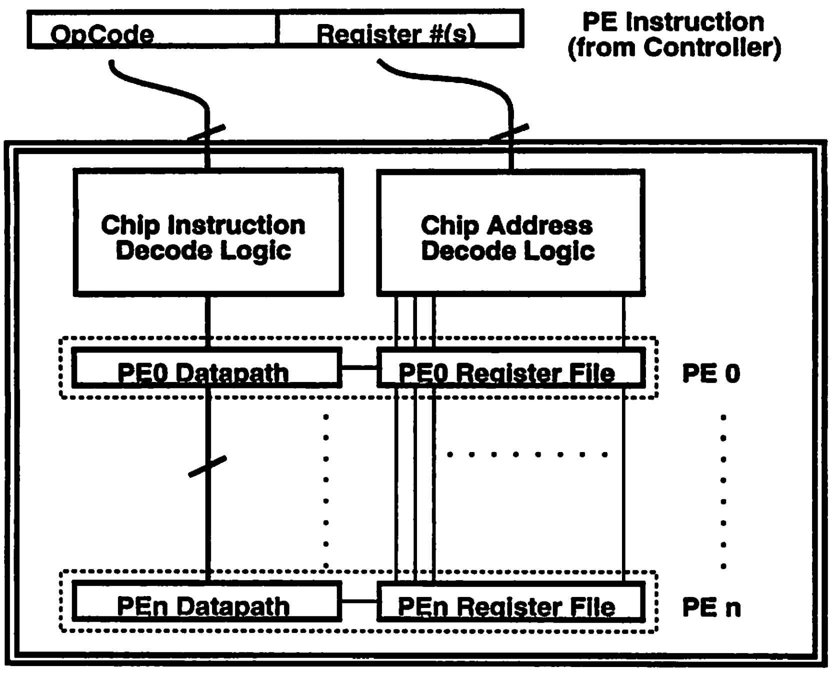

OpCode
Register #(s)
PE Instruction (from Controller)
Chip Instruction Decode Logic
Chip Address Decode Logic
PE0 Datapath
PE0 Register File
PE 0
PEn Datapath
PEn Register File
PE n

Figure 9.1. Shown is a representation of a typical MPA PE chip. Note that there are only single instruction and register address decoders.

- **ADSP-21020** [35] – The ADSP has 80 bytes of on-chip register file, 512K bytes of on-chip cache, plus external memory. Recall that the ADSP is a DSP chip that can be configured to run as a SIMD PE.

- **BLITZEN** [28] – The BLITZEN has 128 bytes on-chip storage; the off-chip size is not known. The on-chip storage is loaded at 4 cycles/bit through dedicated I/O buses.

- **CAAPP** [147] – The CAAPP has 40 bytes of on-chip and 4K bytes of off-chip storage. The on-chip storage is loaded at 5 cycles/bit through a dedicated load/unload mechanism.

- **CM1/2** [140] – The CM1 and CM2 have a flat storage space (not including disk I/O): the 8K bytes of storage are all off-chip.

- **DAP** [4] – The DAP also has a flat storage space: the 128K bytes of storage are all off-chip.

- **DTC** [81] – The Data Transport Computer has 256 bytes on-chip and from 8 to 32K bytes off-chip storage. The on-chip storage is loaded at 16 cycles/bit using the high-speed I/O interface.

- **GAPP** [44] – The GAPP has 16 bytes of on-chip storage, the off-chip storage size is not known. The on-chip storage is loaded at a rate of 132 cycles/bit by bringing the data in from the short side of the 48 × 132 PE modules.

- **MP1/2** [25, 108] – The MP1 and MP2 have 160 bytes of on-chip and 16K bytes of off-chip storage. The on-chip storage is loaded at a rate of 1.14 cycles/bit through dedicated load/unload mechanism. Since the MP1 and MP2 have 4 and 32 bit wide internal datapaths, respectively, accessing off-chip memory is a factor of 4.5 times slower than on-chip for the MP1 and factor of 36.4 times slower for the MP2.

The memory architectures just described can be divided into two categories, flat and hierarchical. The flat memory has the advantages of simplicity of design, since the entire memory can be constructed from commodity parts, and that all the PE chip area can be used for the datapath. A drawback is that the average ALU load latency can be longer than in hierarchical schemes. This consequence does not follow as immediately for MPAs as it does for serial processors, however, since the MPA cycle time has more limiting factors.

In most hierarchical designs, the storage can be partitioned into on-chip and off-chip components. In keeping with our earlier terminology, we refer to on-chip storage as register file, and off-chip storage as main memory, or just memory for short. Historically, the on-chip memory has been very small and has been loaded from the side of the array. As a result, load latencies were on the order of a hundred cycles/bit (see, for example, [79, 117]). Since many important computations require substantial storage, most recent designs have either increased the amount of register file, introduced high-speed register load mechanisms, or both. Also, in most of the hierarchical configurations, the register file can be loaded without interrupting the computation.

We have included the ADSP-21020 as an example of the likely future direction for MPA memory architectures. It is possible, if not likely, that future MPA designs will have a cache level in the memory hierarchy in addition to the register file and main memory. ENPASSANT has the capability of evaluating three level memory designs, as shown in Figure 9.2.

9.2 Evaluating the MPA PE Memory Design Space

The basic problem in evaluating a potential memory hierarchy design in ENPASSANT is that the MVM has a flat memory space with locations specified only by variable name and type, rather than the physical memory and register locations of the target machine architectures. This 'gap' is the cost of being able to evaluate the traces with respect to a large number of target machine register/cache designs without having to regenerate the traces. As a consequence, the physical memory, cache, and register behavior of the program execution must be (re)constructed *a posteriori*.

To see how the reconstruction of register file and cache behavior fit into ENPASSANT, recall the difference between the technique used here—virtual machine emulation and trace compilation—and trace-driven simulation. In the latter method, the instruction set and register

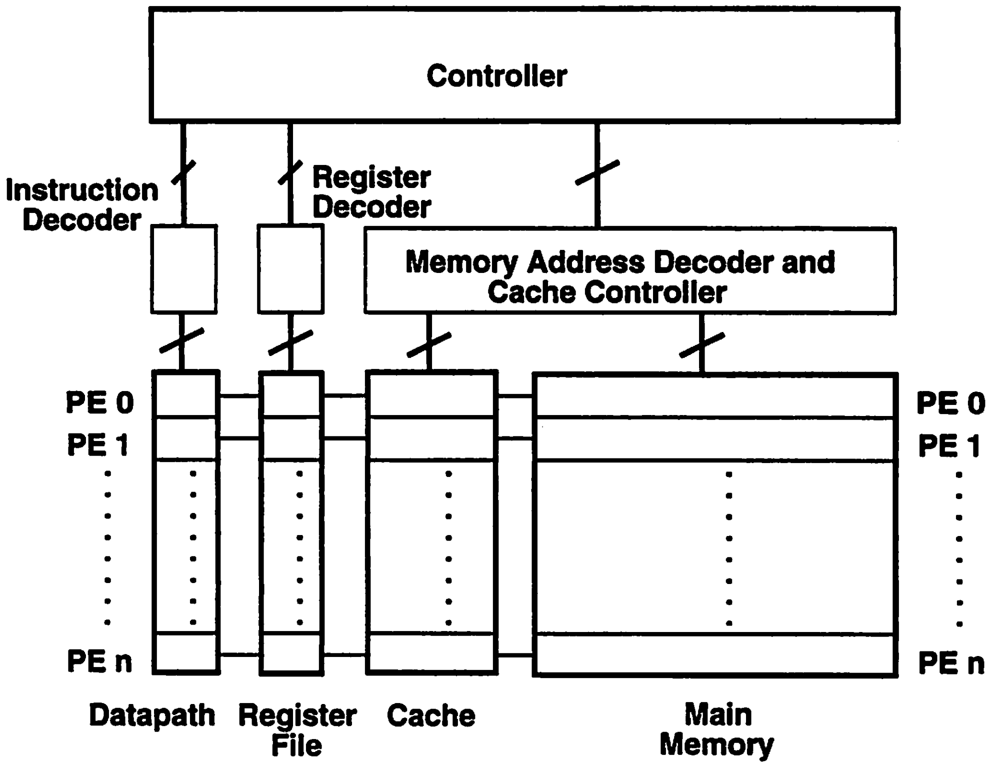


Figure 9.2. Shown is the MPA PE memory design space. Note that all PEs share the same controllers.

architectures are assumed to be known when the trace is generated. Therefore only cache behavior needs to be reconstructed, using well-known methods. In ENPASSANT, the register architecture is not assumed to be known when the trace is generated so register assignment must be reconstructed as well. However, once register assignment has been completed and the memory reference trace generated, the same techniques can be used to analyze the cache as in trace-driven simulation. In particular, ENPASSANT reconstructs register assignment during Pass 5 of trace compilation, while cache behavior is processed after trace compilation has been completed.

9.2.1 Evaluating Register Architectures

In order to evaluate a register architecture it is necessary to determine the number of memory references required during the execution of a test suite program. To determine the number of references requires that the register assignment be reconstructed.

There are two ways that PE registers can be assigned during the execution of a program on an MPA: at compile time or at run time. In run-time assignment, the controller treats the register file as if it were an explicitly managed cache. The controller manages data structures to keep track of which variables are in which registers, and issues load and store operations dynamically as needed. The controller can use any of the standard virtual memory replacement policies. Since keeping track of the register file in software has non-trivial cost, dynamic assignment is a viable option only when the controller is substantially faster than the array. In compile time assignment, the compiler assigns PE registers in the same way that it assigns controller registers and the load and store instructions appear explicitly in the executable image.

We first show how dynamic register assignment is reconstructed and then show that the same basic technique can be used to reconstruct compile-time assignment. The input is the output of Pass 4, the register allocation phase. Pass 4 adds 'pseudo-instructions' to the trace to indicate to the register assignment processor which variables need to be in registers and when.

The critical observation in reconstructing *dynamic* physical memory assignment—whether it be main memory, cache, or register file—is that when we read through the trace sequentially, at any particular record we have exactly the same knowledge that the controller has when it issues the corresponding virtual machine instruction. It follows inductively that we can assign registers using exactly the same technique that the controller would.

Dynamic register assignment techniques involve:

1. checking to see whether a variable that must be in a register at a particular moment is actually there;

2. if it is not in a register, determining whether there is a free register into which it can be loaded; and

3. if there is no free register, then determining, according to some replacement policy, which register should be spilled.

Any assignment policy that the controller can execute at run-time can be reconstructed by ENPASSANT.

The amount of processing required to reconstruct dynamic register assignment depends on the replacement policy and whether register file analysis is to be followed by cache analysis. In particular, if a 'stack-based' replacement policy (for example, Least Recently Used (LRU)) is selected, then the evaluation process can be simplified substantially by using the technique of Mattson, et al. See [106] for definitions and details. What this allows us to do is obtain the number of loads and stores that would be needed *for all size register files* using only a single pass through the trace.

If the trace is to be used for further processing, however, especially if it is to be used for cache analysis, then a memory reference trace must be generated. And if the cache is to be analyzed with respect to the register file architecture—on which its performance critically depends—then memory reference traces must be generated for each candidate register file architecture.

ENPASSANT currently supports several dynamic replacement policies, including, LRU, random replacement, and a hybrid policy that combines evaluation stack allocation with random replacement. The LRU policy spills the register containing the variable whose last access was furthest in the past. The hybrid policy reserves a small number of registers to hold the evaluation stack. The other registers are replaced randomly.

We now examine the effect of register size and dynamic replacement policy on the fraction of time that a set of programs spends on memory references.

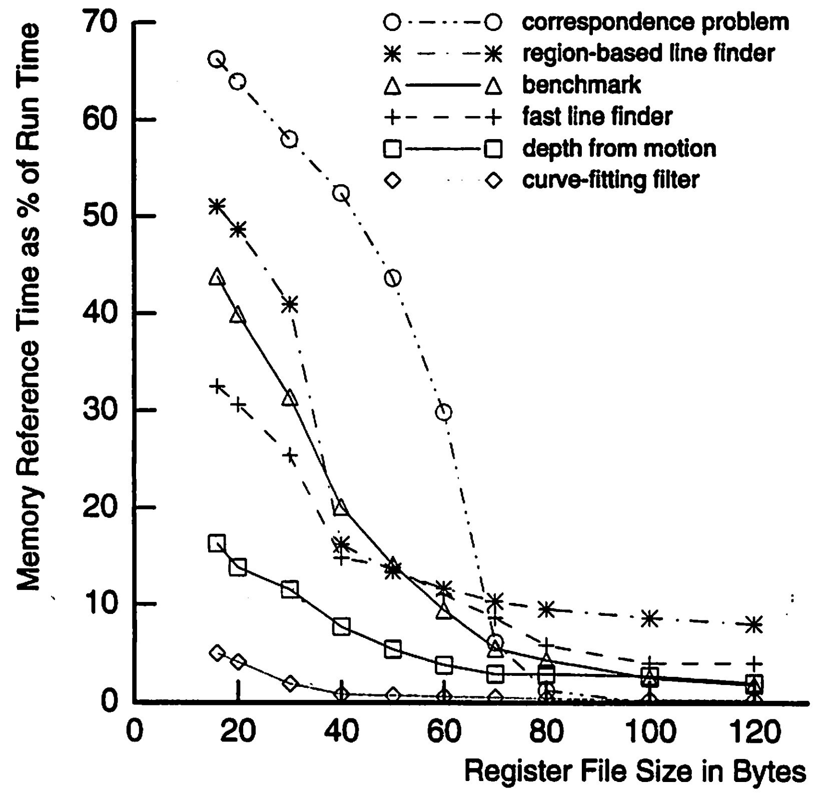


Figure 9.3. Effect of the register file size on the fraction of program execution time spent on memory references. A CAAPP-like model is used.

Figure 9.3 contains the results of six of the benchmark programs using the hybrid replacement policy. A CAAPP-like model was used; however changing the datapath design does not change the shape of the graphs, only the scale of the Y-axis.

The most important observation is that the curves all have distinctive 'knees' (although some are easier to recognize in Figure 9.5). In this respect, the shapes are similar to what one finds when doing memory analysis on serial processors; see, for example, Stone's discussion of memory system design in [131]. These knee shapes in Figure 9.3 are significant because they record a pattern of locality in variable references: in particular they signify a pattern of locality similar to that found in the memory references of serial programs. Serial processor architectures contain features (such as cache) that take advantage of locality of reference to cost-effectively improve the performance of memory access. It follows that if the memory reference patterns are similar, then analogous architectural techniques will also be useful for MPAs.

There are also some observations about Figure 9.3 that confirm expected behavior. One is that the two programs with the highest fraction of floating point computations (depth from motion and curve fitting) are the least affected by changes in register file size. Programs that are compute bound are not affected nearly as much by changes in load/store time. Another observation is that the program with the most homogeneous execution (correspondence problem), the 'tightest inner loop,' also has the most well-defined working set. Either all the variables are assigned to registers at the beginning of the loop, or they are not.

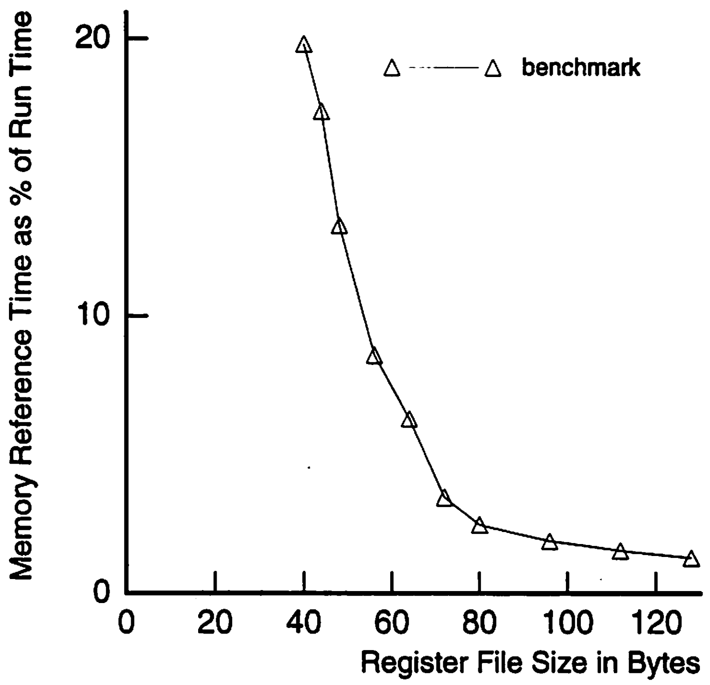


Figure 9.4. Detail Figure 9.3. Note that at this resolution the location of the knee can be more tightly bounded.

A final observation is that the memory evaluator is sensitive enough to capture fine detail in the memory reference patterns. In Figure 9.4 we present in detail of one of the curves in Figure 9.3. By 'blowing up' part of the curve, we notice the precise location of the knee of the curve. This level of detail may very well be critical in systems where the cost of memory references is higher than it is for the CAAPP-like model shown. This scenario is likely if the trend continues that the number of PEs on a chip increases more quickly than the chip I/O capability.

We now examine the effects of two dynamic register replacement policies on the time spent performing memory references. These two policies are Least Recently Used (LRU) and a hybrid policy that combines the reservation of registers for the evaluation stack with random replacement. We observe in Figure 9.5 that the choice of register replacement policy has two principal consequences. The first is that LRU has slightly better performance than the hybrid policy.

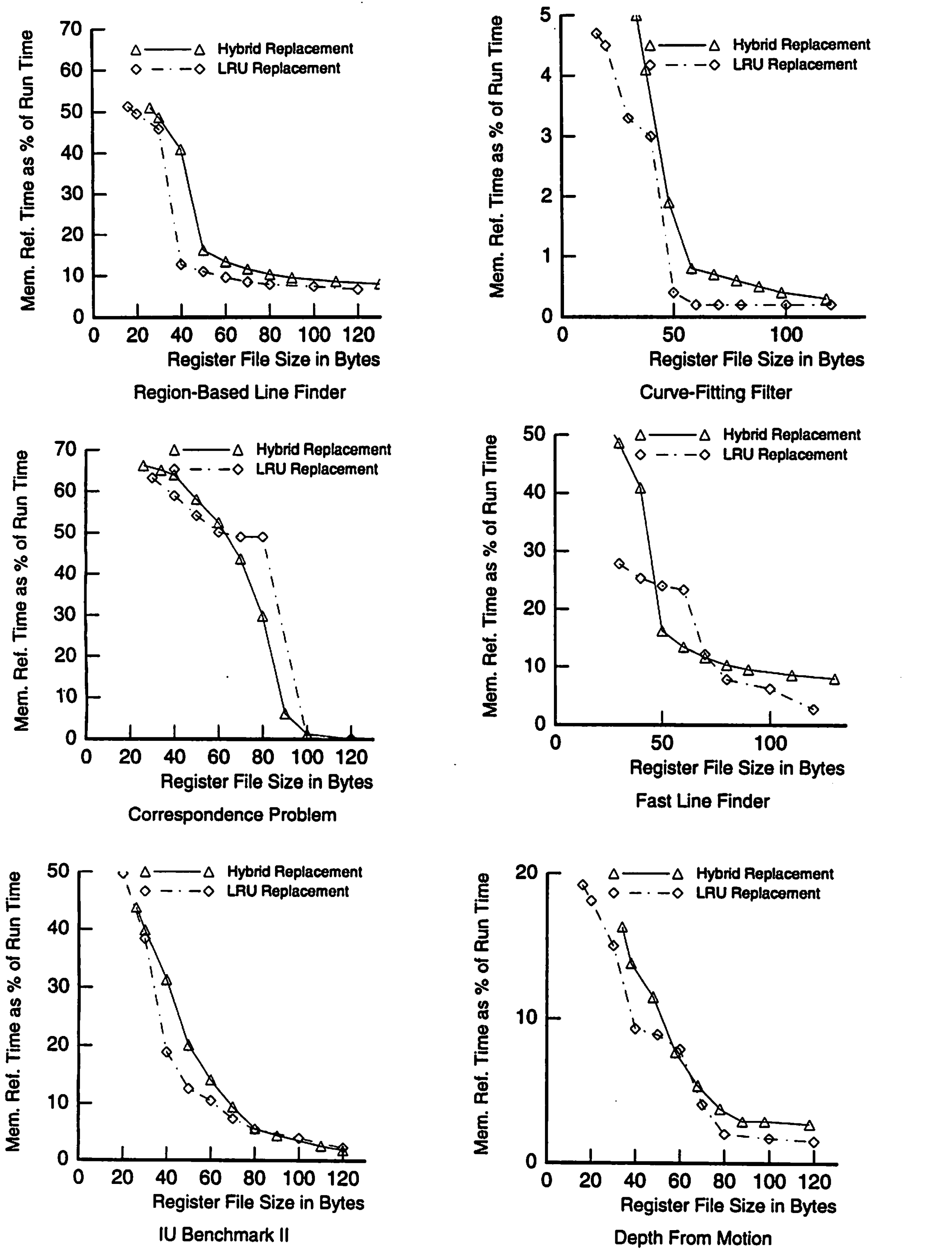


Figure 9.5. Effect of dynamic register replacement policy on the total cost of memory references for several test suite programs.

This is to be expected since LRU uses significant run-time information to decide which variable should be swapped out. The second difference is that memory reference patterns caused by the hybrid policy are smoother. This is also expected since it indicates that working sets get swapped gradually, rather than all at once.

By far the most important result in Figure 9.5, however, is that the effect of the register replacement policy on memory performance is relatively small when compared to, say, the difference in performance across programs shown in Figure 9.3. Although the LRU curves are generally to the left of the hybrid curves, the pairs have the same basic shape (except for a few bumps). This is especially true of the most complex programs, the IU Benchmark and the region-based line finder.

The significance of the result lies in its implication on the reconstruction of compile-time register assignment. The hybrid policy, blindly assigning registers to the evaluation stack and randomly assigning the rest, should give worse performance than the policy of a reasonable compiler. LRU, as it uses dynamic information about locality of references, should give better performance. The policies, as a pair, are thus likely to provide bounds on the possible effect of compile-time register assignment on register file performance. Since the spread is relatively small it follows that either of these policies can be used to approximate compile-time register assignment.

9.2.2 Evaluating Cache Architectures

Evaluating cache architectures from a memory reference trace is straightforward and uses the same basic technique that was used for reconstructing dynamic register assignment. The cache evaluator in ENPASSANT allows cache architectures to be evaluated with respect to cache size, line size, and associativity.

In Figure 9.6 we see how the cache analyzer can be used to evaluate the relative benefits of associative versus direct-mapped caches. Shown is the effect of associativity on the relationship between the cache size and the hit rate. Six of the benchmark programs were run on the CAAPP-like model with a virtualization factor of 4. The register file size was 30 bytes.

Although these results are not sufficient for use in final design decisions—the effect of the cache design within the entire memory architecture must be examined for that to be possible—they useful in showing what those effects are likely to be. To make caching cost-effective, high hit rates are essential. In particular, we see that whereas an associative cache having a size of 20 blocks is sufficient to consistently achieve hit rates over 95%, a direct-mapped cache must be twice that size to achieve the same performance. The exceptions are the programs that require only a small amount of memory (curve-fitting filter, correspondence problem).

The cache analyzer also allows the user to evaluate the effect of block size on cache performance. Such a series of experiments is shown in Figure 9.7. All programs were again run on a CAAPP-like model with a virtualization factor of 4. Here we see a result that is completely

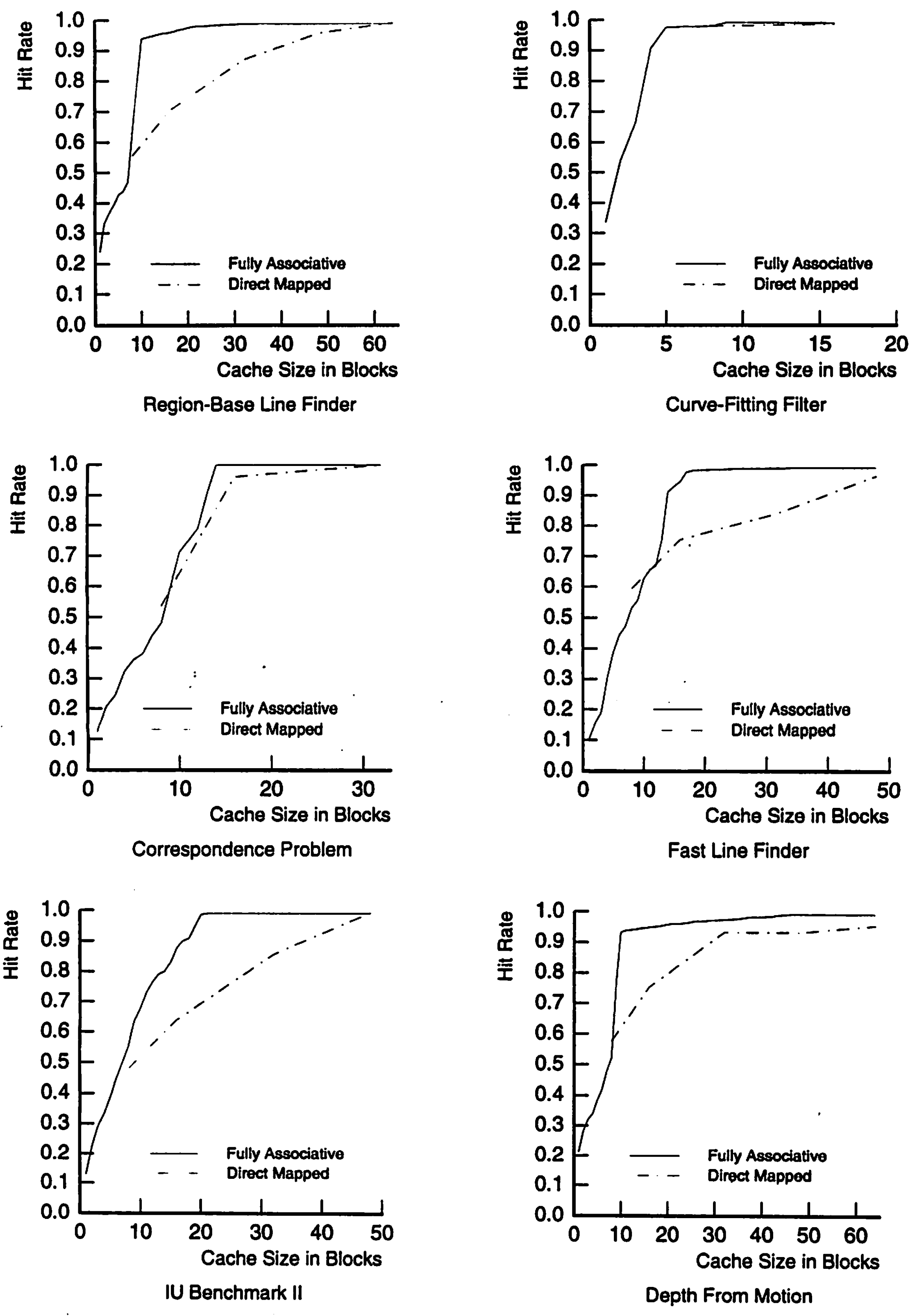


Figure 9.6. Comparision of fully associative and direct mapped cache performance.

counter to what one would find in a similar analysis of serial processor code: the hit rate *decreases monotonically* as the block size increases.

To understand what is going on here, let us first repeat the standard reasoning as to why we expect the hit rate to first increase and then decrease with block size. Most programs written for serial processors have been found to exhibit spatial locality in their memory references. That is, variables referenced near each other during the execution of a program are also likely to reside near each other spatially in memory. The closer a variable is to the variable just referenced, the likelier that the variable will be referenced next. Therefore, the larger the block that is brought into cache on a miss, the longer the expected time until the next miss. This phenomenon is limited by the fact that an increase in block size means a decrease in the number of blocks in cache, assuming we have not varied the size as well. Thus when the block size becomes too large, we lose much of the performance gained by taking advantage of temporal locality. See, for example [69] for details.

As we see, however, the results in Figure 9.7 do not show this effect. The explanation is that there is much less spatial locality in the memory references in the MPA test suite programs than in serial programs. One reason for this has been alluded to earlier: much of the spatial locality in serial machine programs comes from stepping through arrays. In MPAs we are already processing arrays two dimensions at a time.

9.3 Chapter Summary

In this chapter we have presented the memory hierarchy evaluation techniques used by ENPASSANT. As far as we are aware, these are the first published results of working set sizes and locality effects for MPAs. The working sets for the test suite programs running with a virtualization factor of 1 (as indicated by the knees in the graphs in Figure 9.3) are between 25 and 80 bytes. Cache associativity was found to be important: direct mapped caches must be more than double the size of fully associative caches to achieve similar performance. Also, we found that the test suite programs exhibit little spatial locality in their memory references and so very small cache block sizes are favored.

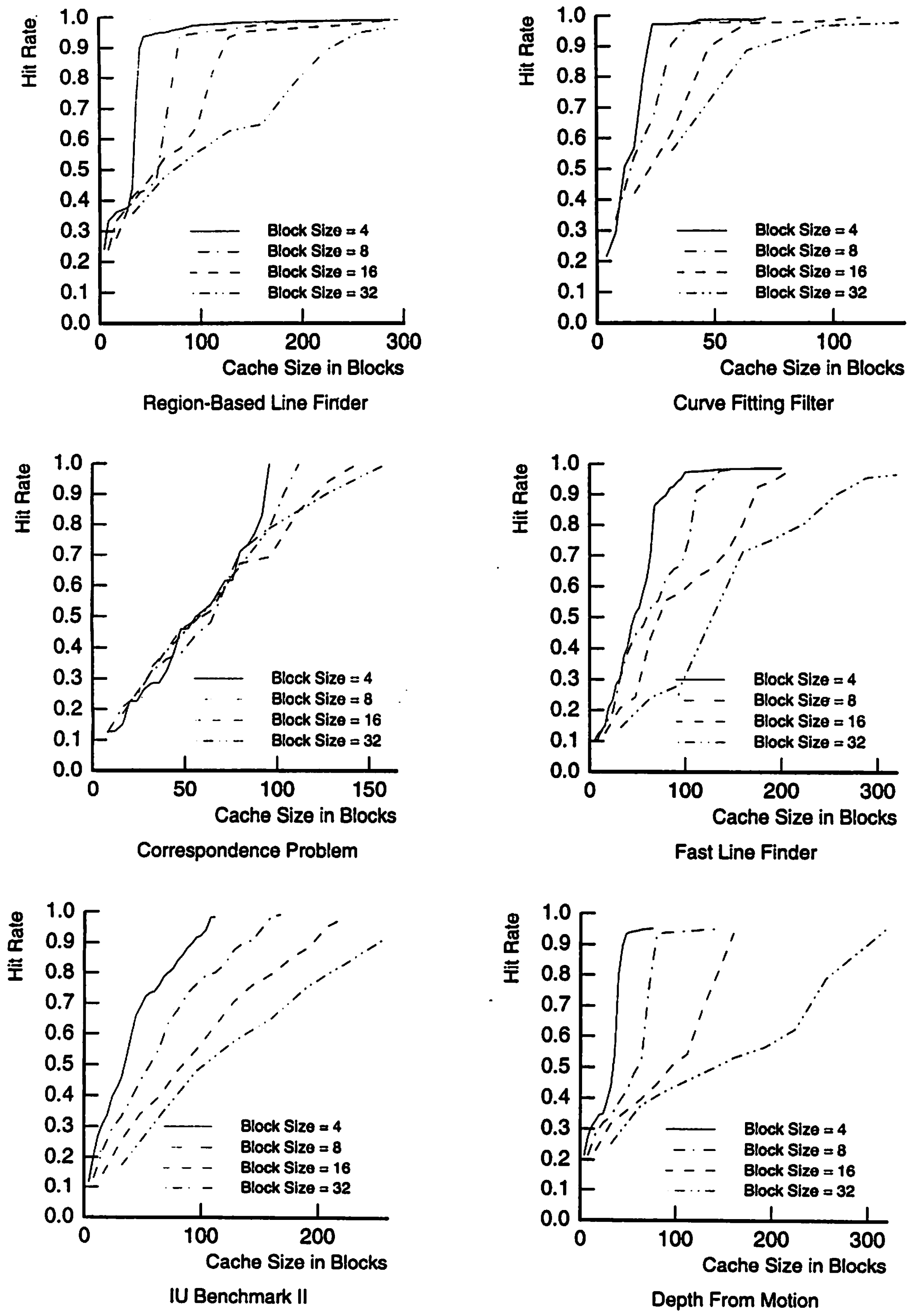


Figure 9.7. Effect of varying the cache block size on the relationship between the hit rate and the overall cache size. The cache is fully associative.

CHAPTER 10

COMMUNICATION NETWORKS

In this chapter we examine the evaluation of MPA communication networks, or more precisely, we examine the evaluation of communication operations and how their performance depends on the network. We make the distinction between evaluating operations and comparing networks because of the differences in functionality of the various communication mechanisms within the MPA domain. Recall that while some networks—that of the CM2, for example—are capable of implementing most of the MVM communication operations directly, many of the other networks cannot. As a consequence, MPAs in the latter situation must emulate many of the MVM communication operations using the communication features that are available.

We begin this chapter with a discussion of the issues in communication network evaluation. There follows a review of communication networks on existing machines and how we approximate that design space in ENPASSANT. We then present the algorithms used by the various networks to emulate the MVM communication instructions. We close this chapter with some some sample results showing architectural dependencies of MVM communication instruction performance.

10.1 Issues in Communication Network Evaluation

Evaluating MPA communication, just like evaluating the MPA datapath and memory, has its particular challenges. One is the wide variety of networks: though the class of MPAs has been defined so that all its members have nearest-neighbor mesh networks (or the ability to emulate one efficiently), many MPAs have additional broadcast, packet switched, or circuit switched networks.

Another difficulty in evaluating communications is that it can be extremely time-consuming. For example, the execution of an MVM communication primitive on a CAAPP-like model can translate into an emulation of 100,000 target machine (though many fewer MVM) instructions. In this case, though, the time of evaluation is still roughly proportional to the time of execution on the target machine. A more costly situation (in evaluation time) is when a communication network must be simulated, at similar expense, to emulate only a single MVM instruction. This is the situation that arises when a network with data-dependent performance executes a communication operation. In this case the evaluation time can be significantly greater, proportionally, than the target machine execution time.

The biggest problem, however, is not only that communication operations can be very time consuming to evaluate, but that the context (data and architectural specification) needed to perform the evaluation is storage intensive. For example, the context needed to evaluate a particular MVM +Route operation includes not just the pertinent parts of the architectural specification, the virtualization factor, and the type of the data being routed—factors important for most MVM operations—but the destination vector as well. When the size of this last quantity (which can consist of 256K bytes of data) is compared with the few bytes needed to record most instruction executions, we see that communication instructions require a special evaluation mechanism.

As noted above, target machine communication networks must be evaluated by how long they take to execute particular instructions. We partition the routing instruction/network pairs into three categories which have different evaluation requirements.

- The first set of instruction/network combinations is not data dependent and is easy to compute. Included in this category are all nearest-neighbor moves, broadcast operations on broadcast networks, and scan operations on packet switched combining networks. These combinations are no more difficult to evaluate than any of the instructions we examined in the datapath evaluation section and so are handled in the MVM/target machine dictionary generator.

- The second set of instruction/network combinations are not data dependent, but the architectural dependency is complex. In order to evaluate these combinations, one, but only one, emulation or simulation must be run to model the performance of the combination. This category includes scan operations on broadcast and nearest-neighbor networks.

- The final set of instruction/network combinations are data dependent. In order to precisely evaluate these combinations, they must be emulated or simulated every time they appear. Examples include all Route instructions where the destination vector is not known *a priori*, and region operations on meshes and packet routing networks.

There are at least three possible techniques for handling data dependent communication instructions, all of which have their benefits:

1. We can evaluate the instruction during virtual machine emulation. This technique permits precise evaluation, but has the drawback that the virtual machine emulation must be rerun in its entirety every time the network or the array size is varied. Although this is still much faster than detailed simulation, but is nonetheless quite costly.

2. We can record the data on which the timing of the communication instruction depends and store it along with the rest of the trace. The problem with this approach is that size of one context can be hundreds of Kbytes, making the storage it requires equivalent to that of several thousand MVM instructions. In particular, we are then limited to storing just a few hundred such instructions.

3. We can separate out communication evaluation from the rest of the evaluation process. That is, for each test suite program, we can evaluate the performance of communication operations with respect to the candidate networks, ignoring the rest of the virtual machine emulation. If the variability of the network performance is small, then the values can be included in the architectural specification.

A final comment: although evaluating the MPA support for data dependent interPE communication operations is complex, this is not unexpected. Either the amount of time required to evaluate those operations is proportional to the amount of time they take to execute on the target machine, or the target machine has very sophisticated communication support rivaling the array itself in complexity.

10.2 Current MPA Communication Networks

In this section, we present 1) the current state of the MPA communication network design space, 2) some possible areas into which the space is likely to expand, and 3) the abstract space that we use in evaluating the current and future design spaces.

As with the other two architectural components we examine in detail (memory and datapath), the MPA communication network architecture is constrained by SIMD control. In particular, all communication operations are synchronous in the following sense.

1. Communication operations are instructions initiated by the controller through broadcast to the array (just like any other).

2. All PEs taking part in the communication operation begin processing at the same time.

3. The communication operation ends when all data have reached their destinations.

4. During communication, any particular PE is either taking part in processing the instruction or idle. There are some MPA configurations where datapath instructions can be executed during communication, but whether a PE takes part or not does not depend on whether the PE is taking part in the communication operation.

5. The network cannot be used for another communication operation until the one running has been completed.

See Figure 10.1 for a common MPA configuration with a nearest neighbor network and also a second unspecified communication network.

We begin by describing the communication architectures of some typical and well-known MPAs. We again restrict our consideration to machines that have actually been built, and that are relatively recent.

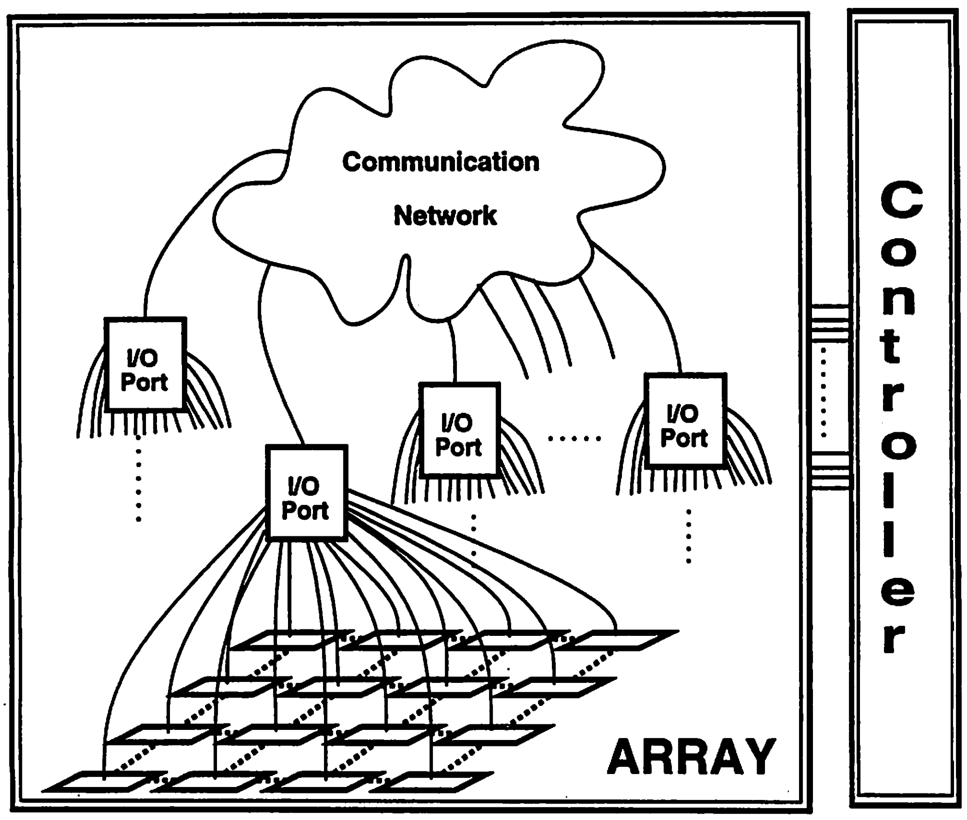


Figure 10.1. In a typical MPA communication scheme, each PE has connections to both nearest-neighbors and to a dedicated router network.

- **BLITZEN** [28] – The BLITZEN interPE communication mechanism is the X-grid nearest-neighbor routing network. It is similar to the routing networks of the MPP [117] and many other MPAs, except that communication with PEs in the NE, SE, SW, and NW directions are just as efficient as communication in the N, E, W, and S (NEWS) directions. See Figure 10.2. To communicate, the controller directs PEs to put data into the communication I/O buffer, whence it is sent to the corresponding buffer in the specified neighbor.

- **CAAPP** [147] – The CAAPP has two interPE communication networks: a nearest-neighbor network and a Coterie network. The nearest-neighbor network is similar to the BLITZEN network, except that direct communication is only supported in the four NEWS directions and that no intermediate I/O buffer is required. In the CAAPP, a PE requires no more time to read from the register file of an adjacent PE than it does to read from its own.

  The Coterie network is a member of the class of networks often referred to as reconfigurable broadcast meshes. The basis for all these networks is a bus to which all PEs are connected via a network port. During operation, PEs write to their network ports, idle for the number

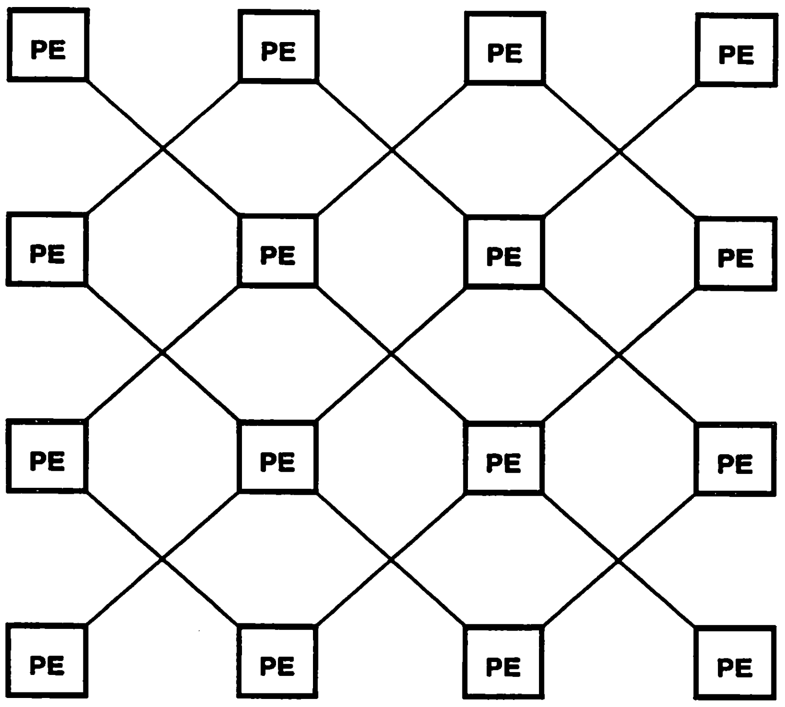


Figure 10.2. A 4 × 4 X-grid network.

of cycles necessary for the signal to propagate, and then read their ports for the result. If multiple PEs write to the Coterie network at the same time, the result is the OR of the signals put onto the network. What really makes the Coterie network useful, however, is that PEs control switches that disconnect their local part of the network. Thus, under program control, the coterie network can be reconfigured into any partition of a mesh. The local buses, or Coteries, can then be used for further processing. See Figure 10.3 for an example.

- **CM2** [140, 7, 16] – The CM2 has two interPE communication networks, one that implements local communications among PEs in local groups of 16, and one that implements global communication among those local groups, or nodes. Each local, or 'flipper,' network consists of a 4 stage butterfly network which is capable of delivering permutations within the local group.

  The global communication network consists of 4096 router nodes (one for every 16 PEs) that are interconnected in a 12 dimensional hypercube. Packets are transferred between the PEs and their associated router nodes via injector and ejector circuits. The router nodes are packet switches with circuitry to receive, send, buffer, route, and combine packets. The router nodes transfer packets in petty cycles—which in turn are composed of 12 dimension cycles—until routing has been completed. During the *i*th dimension cycle, each router node

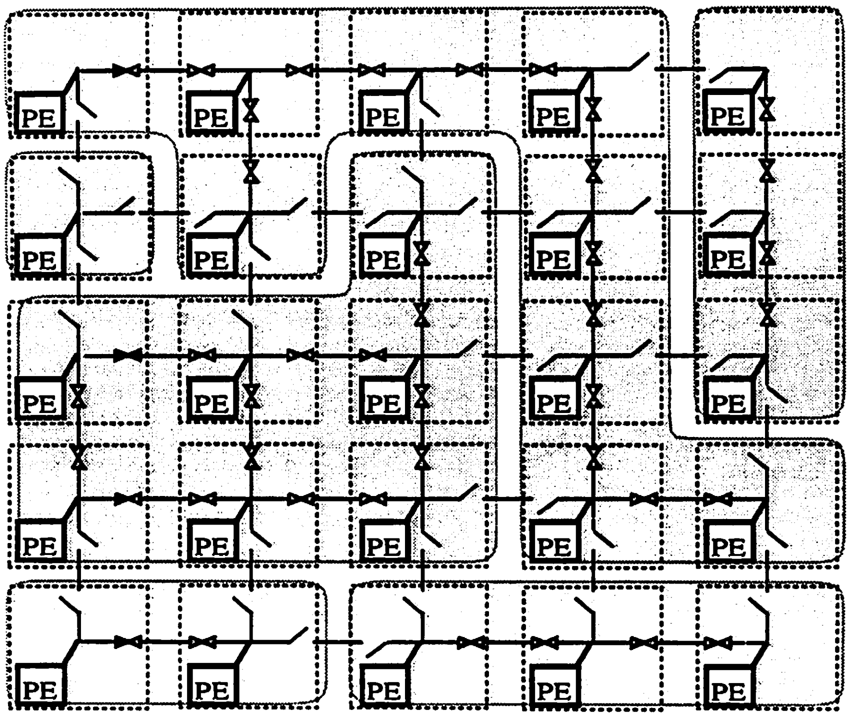


Figure 10.3. A 5 × 5 coterie network with switches shown in arbitrary settings. Shaded areas denote coteries (the sets of PEs sharing the same circuit). Dashed boxes circumscribe the switches over which each PE has control.

selects a packet which needs to be transferred in the *i*th direction, and sends it to the associated router node. More details of this network are presented in Section 10.3.3.

By combining operation of the local and global networks, the CM2 has the capability of implementing nearest neighbor moves in any of the 12 hypercube dimensions. With proper embeddings, a conventional nearest-neighbor mesh can be emulated.

- **DAP** [4] – The DAP has two interPE communication networks, a nearest neighbor mesh with buffering similar to that in the MPP, and a broadcast network. In the DAP broadcast network, each PE has access to a row and to a column bus, rather than the entire array as in the CAAPP. Also, the network is fixed; that is, there are no switches for reconfiguration. See Figure 10.4.

- **DTC** [81] – The DTC has a nearest-neighbor network that can be used to transfer data in either 2 dimensions or 3 dimensions with standard buffering.

- **GAPP** [44] – The GAPP has a two dimensional nearest-neighbor (NEWS) network with standard buffering. See Figure 10.5.

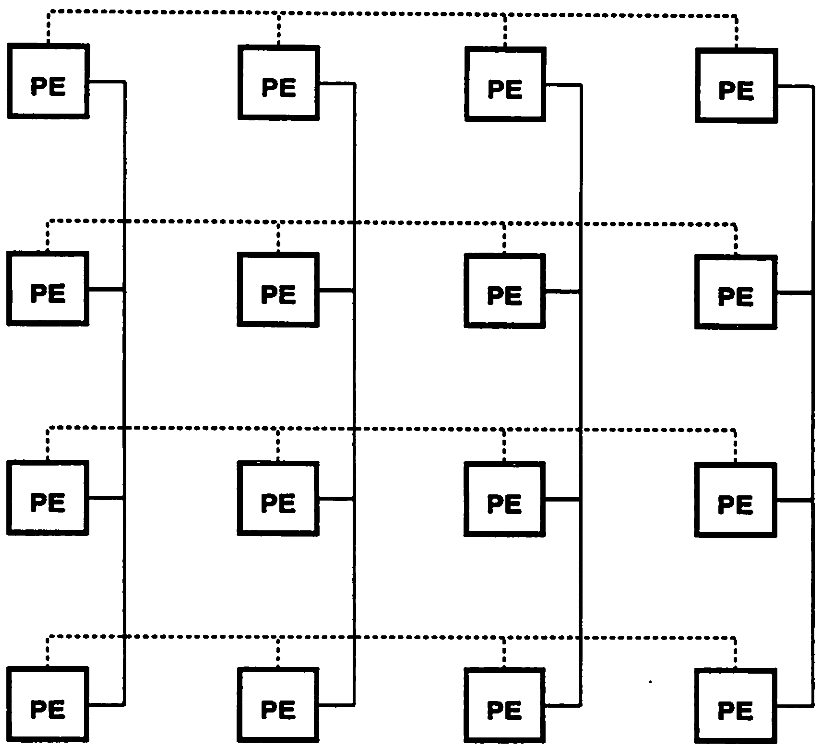


Figure 10.4. A 4 × 4 network of horizontal and vertical buses.

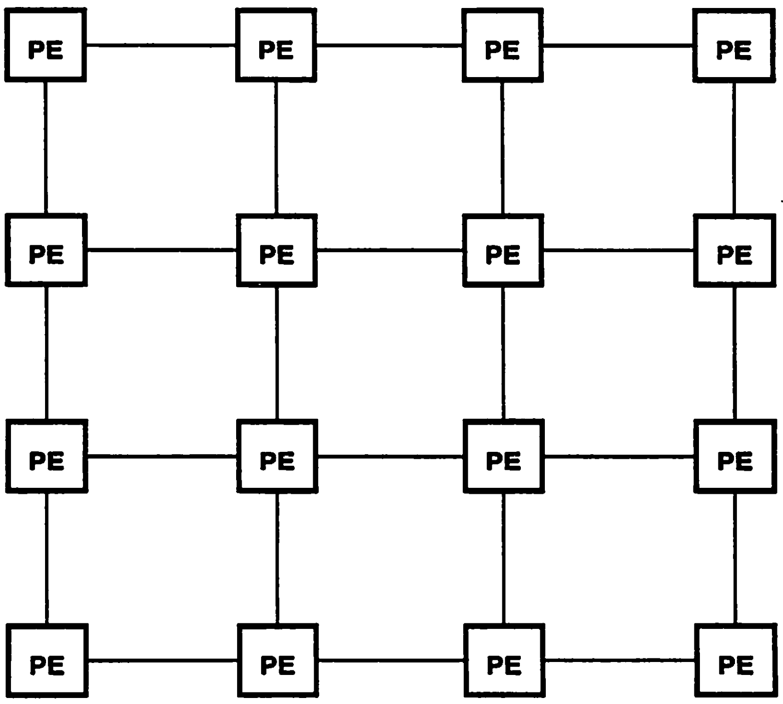


Figure 10.5. A 4 × 4 mesh network.

- **MP1/2** [65, 25, 108] – The MP1 and MP2 have identical communication systems, consisting of two networks. The first is an X network similar to the one in the BLITZEN, the second is a circuit switched permutation network.

  The circuit switched network is a self-routing three stage Benes/Clos network. The network has 1024 input and output ports, or one for every 16 PEs. When communication is initiated, PEs send their packets to the network I/O port, whence the packets are sent through the switches. At each switch, the appropriate address bits are stripped from the packet header and used to route the packet to the next switch. The entire path is held open until the packet has reached its destination and the destination has returned a confirmation signal. Packets that have not reached their destinations at the end of a routing cycle are lost and must be transmitted. Retransmission continues until all packets have reached their destinations. Because of the ratio of PEs to routing ports, the lack of intermediate buffering, and the nature of the network, many permutations require multiple passes to complete. More details of this network are presented in Section 10.3.2.

All of the MPAs have nearest neighbor networks; this is a defining characteristic of the class. The other types of networks can be categorized as broadcast, packet switched, and circuit switched.

10.3 The MPA Communication Network Design Space

In this section we consider the current and future design spaces for the four network combinations and how we model/simulate them.

10.3.1 Mesh and Broadcast Networks

Current mesh networks have the following parameters.

- They can require a time to set up communication (set-up time), or not. In the former case, it is usually possible to amortize the set-up time over multiple transfers.
- The transfer latency is one or more cycles per transfer.
- The interPE datapath width is one bit wide.

It is possible that this last value could vary in the future, depending on scenarios involving advances in packaging technology and how architects decide to use them. There are at least four scenarios where the interPE bandwidth will probably increase.

1. If architects decide that there are advantages to putting fewer, more powerful processor on each chip, then there will be more pins available for interPE communication.
2. If packages gain pins at a faster rate than chips gain circuit capacity, then there will again be more pins available for interPE communication.

3. If MPAs are constructed using multichip modules, then more inter-chip (and therefore more interPE) bandwidth will be available.

4. If the entire array can be constructed on a single chip, then an interPE path width equal to the internal datapath width is not unreasonable.

Broadcast networks have similar parameters to mesh networks: they require a certain amount of set-up time, the transfer latency is some number of cycles per transfer, and the interPE bandwidth is one bit per transfer per PE. Since the transmission hardware for such a network is similar to that of a nearest-neighbor network, there is the possibility that the bandwidth will increase here as well.

All of the parameters, for both nearest-neighbor and broadcast networks, are data independent and so can be included in the architectural specification that is input to the MVM/target machine translator.

10.3.2 Circuit Switched Networks

10.3.2.1 Definitions and Properties

The MP1/2 circuit switched network belongs to a family of routing networks often referred to as multi-stage interconnection networks (MINs). The function of these networks is to route permutations: that is, to send a set of packets—each packet with a unique destination tag—to the correct outputs. MINs are all variations of cross-bar networks whose representatives trade off latency, completeness of permutation set, switch complexity, and number of stages. Some particular members of the MIN family are as follows. We assume N inputs and outputs.

- **Crossbars.** They contain a single stage with a single $N \times N$ switch. Crossbars route all permutations without blocking. The problem is that the switch has N^2 complexity and so is impractical for very large networks.

- **Clos Networks [43].** Clos made the discovery that it was possible to trade off speed for switch simplicity. Clos networks have 3 stages, but the switch complexity is only $N^{3/2}$.

- **Benes Networks [21].** Benes showed that the lower bound on switch complexity to route all permutations is $2N \log N$. Such networks have $2 \log N$ stages, each containing $N/2$ 2×2 switches. Although these networks are non-blocking, paths must be calculated off-line to route some permutations.

- **Log-Depth Networks.** The minimum switch complexity to be able to route a packet from any input to any output is $N \log N$. One configuration of a Log-Depth Network has $\log N$ stages, each containing $N/2$ 2×2 switches. A large number of these networks have been examined, including the butterfly, cube-connected-cycles, deBruijn, base-line, delta, omega, etc. See [92] for a survey of their properties. What they have in common, however, is that they do not route many of the N^N possible permutations without blocking.

MINs with N input and output ports (as shown in Figure 10.6) can have from 1 to $\log N$ stages, each with N input and output connections. Each stage can have from 1 to $N/2$ switches, each of which is a cross-bar with a size that ranges from $N \times N$ to 2×2. The switch size depends on the number of switches in that stage and whether there are redundant paths to route packets that would otherwise collide. With redundancy the switch size is, in theory, unbounded, although in practice the size is limited to the maximum size cross-bar switch that can be placed on a chip. There are also more complex MIN configurations, but they have not found their way into MPA designs and so will not be discussed here. See [37] for a survey.

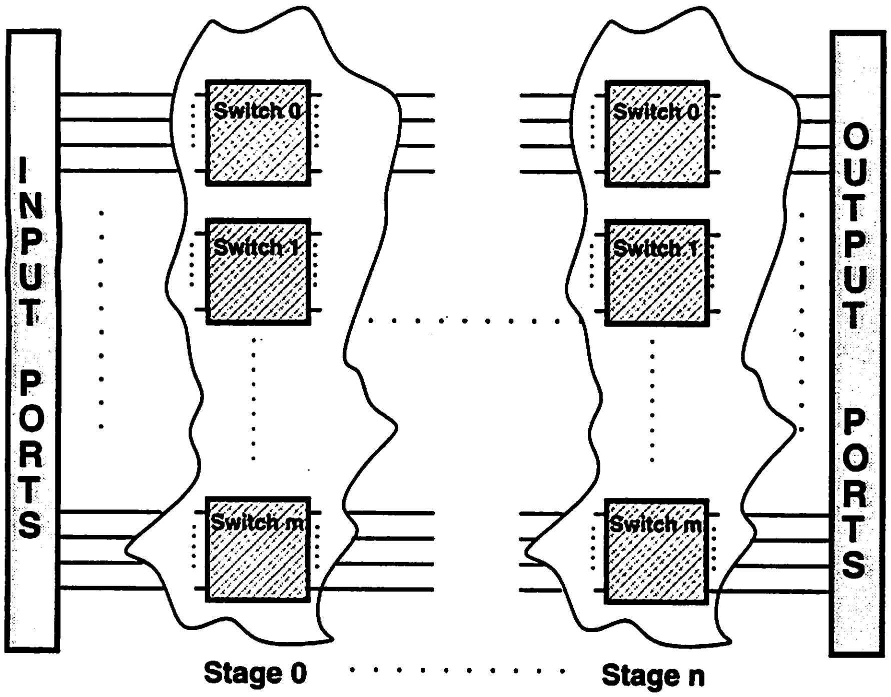


Figure 10.6. Shown is a typical multi-stage interconnection network (MIN).

MINs are specified as follows. Let there be N inputs and N outputs. There must then be $\log N$ address or tag bits. During each stage, some number of tag bits gets decoded; in a cross-bar network, all $\log N$ tag bits get decoded in the single stage. In Log-Depth networks, each of the $\log N$ stages decodes only a single bit. Any switch size in between is also legal. The number of switches required for a stage where i bits are decoded is $N/2^i$. The switches have a minimum size of $2^i \times 2^i$ with larger size desirable to reduce the amount of packet loss.

In the event that a network is anything less than completely non-blocking (all except cross-bar and Clos networks), there is a significant chance that packets will collide. Since MPA MIN networks must be self-routing to be cost-effective, this is true of Benes as well as Log-Depth networks. Since buffering mechanisms are very expensive, MPA MIN networks have taken the approach that when two packets collide, one will be transmitted and the other lost. The fact that lost packets are a common occurrence has two implications: the first is that its arrival at a receiver must be confirmed to the sender; the second is that a routing operation often requires multiple passes.

We now show the benefits of using larger switches than necessary to increase the number of packets routed per pass. If a stage uses 2×2 switches, then for each pair of packets that enters a switch, there is a 50% chance that one will be eliminated due to a collision. If a 4×4 switch is used instead of two 2×2 switches, then the probability of 2 out of 4 packets being blocked is 1/8, the probability of 1 out of 4 packets being blocked is 1/2, and the probability of no packets being blocked is 3/8. Thus the expected percentage of packets remaining after passing through the stage of 2×2 switches is 50% while that for a stage of 4×4 switches is 62.5%.

10.3.2.2 The Circuit-Switched Network Simulator

The parameters of the ENPASSANT circuit switched network simulator are as follows.

- **Number of Network I/O Links.** Networks generally do not have direct links to every PE, rather there are commonly only one or two network links to each PE chip. This value also gives the number of PEs sharing a link. The logarithm of the number of links gives the number of tag bits.

- **Injector/Ejector Latency.** ENPASSANT assumes that there is injector and ejector circuitry between the PEs and the I/O ports. The injectors queue packets waiting to enter the network while the ejectors direct arriving packets to the correct PE.

- **Number of Stages.** This value can vary from 1 to $\log N$.

- **Topology.** The topology specifies how the wires are connected between switches in consecutive stages. ENPASSANT supports the baseline, butterfly, and omega networks. See [92] for their definitions.

- **Stage Parameters.** Each stage of the network is specified by the number of bits decoded, size of the switches, and the latency.

Figure 10.7 shows a comparison of the performance of the MP1 router network as measured by Prechelt [119] and the ENPASSANT MIN simulator when modeling that network. The number of passes through the network required to complete the route is plotted with respect to the number of senders for random patterns. The simulation results are the average of 10 trials for each point; the error bars represent the standard deviation of the average. The ENPASSANT MIN simulator models the MP1 network with very high confidence.

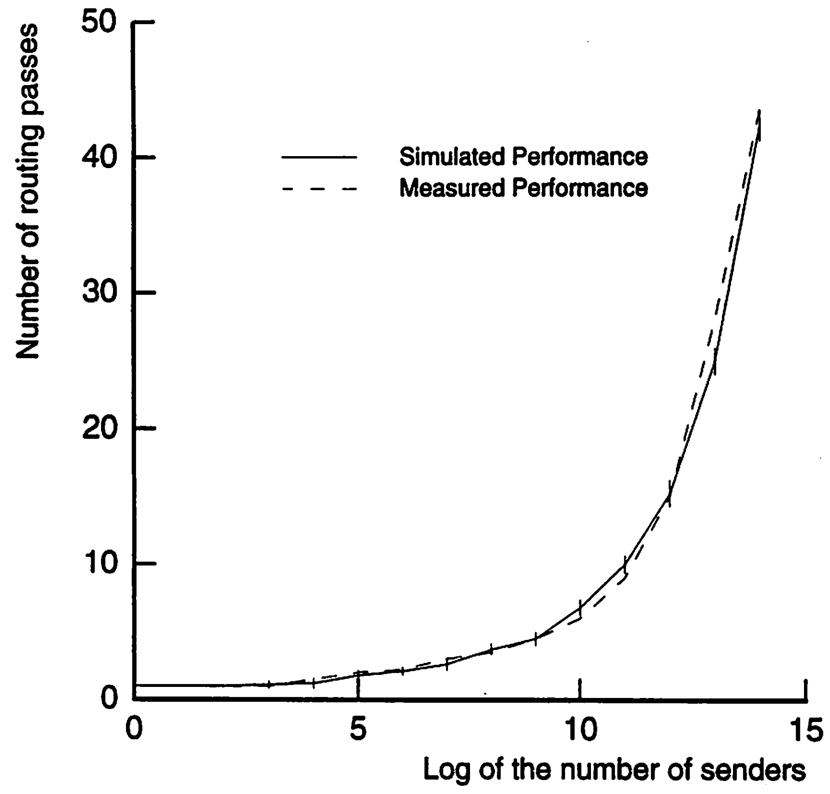


Figure 10.7. Number of passes required to route random patterns of various densities on the MP1 router network as obtained by measurement and by simulation using ENPASSANT.

10.3.2.3 A Circuit-Switched Network Case Study

We now show a case study of how ENPASSANT can be used to help evaluate circuit switched network designs. We compare two networks from the same family of processors having somewhat different designs.

The DEC MPP router network [65] was to have 4096 inputs and outputs and three stages. Since log(4096) = 12, that many tag bits are required to route each packet. Each stage decodes 4 tag bits. Only the last stage, however, has the minimum size 16 × 16 switches; stages 1 and 2 have 64 × 64 switches. The switches in stages 1 and 2 thus allow up to 4 packets that map to the same output to be passed on to the next stage. The total switch count is 128 64 × 64 and 256 16 × 16 switches.

The MasPar MP1/2 network [108] has 1024 inputs and outputs and three stages. Although each stage uses 64 × 64 switches, the first two stages decode only 2 bits apiece, while the last stage decodes 6 bits. The reason for this configuration is that each board has 64 clusters of 16 PEs, each of which shares one I/O port. Thus each board has three 64 × 64 switches and the number of wires running through the backplane is significantly reduced over that required by the DEC MPP design. The total switch count is 48 64 × 64 switches.

After running simulations of random permutations, we find that the MasPar MP1/2 network requires more than 40 passes to complete a route, while the DEC MPP network requires less than a third that many. Although the cost-benefit is not always the same as cost-performance, the DEC MPP network requires far more than three times the number of wires and switches to achieve three times the performance of the MasPar MP1/2 network.

10.3.3 Packet Switched Networks

10.3.3.1 Definitions and Properties

The CM2 routing network belongs to the family of packet switched interconnection networks (PINs). PINs are similar to MINs in that packets are self-routing and that a number of PEs can share input and output ports. PINs therefore also require injector and ejector circuitry to queue input packets and route output packets. There are many differences, however.

- Packets always queue at intermediate nodes. If a packet arrives at a node and there are already packets there, then the packet may be delayed for some number of routing cycles.
- Since there is a queuing mechanism, packets are not lost on collisions. If a queue is filled, then packets are misrouted rather than lost.
- Switches are interconnected via static rather than dynamic topologies. This somewhat artificial distinction (see [7] for a discussion) implies that the switches are associated with clusters of PEs, rather than being independent of the PE array. Thus packets routed in PIN networks must only travel as far as the node associated with the destination cluster, in contrast to MIN networks where packets must always traverse the entire the diameter of the network.
- The family of interconnection topologies is also different. PIN topologies in MPAs are generally k-ary n-cubes where n refers to the number of dimensions and k refers to the number of switches in each dimension. Two examples are the k-ary 2-cube, or mesh with k nodes on a side, and the 2-ary (binary) n-cube, or the n-dimensional hypercube.

PINs and MINs have very different performance properties. MINs route packets through switches at a speed that is an order of magnitude higher than that of PIN networks. However, PIN networks can take advantage of locality in the communication pattern: if all packets are only traveling a short distance, the difference in switching time can be overcome. Also, PIN networks with combining hardware can be used to route reductions and scans as well as permutations (see, e.g. [26]).

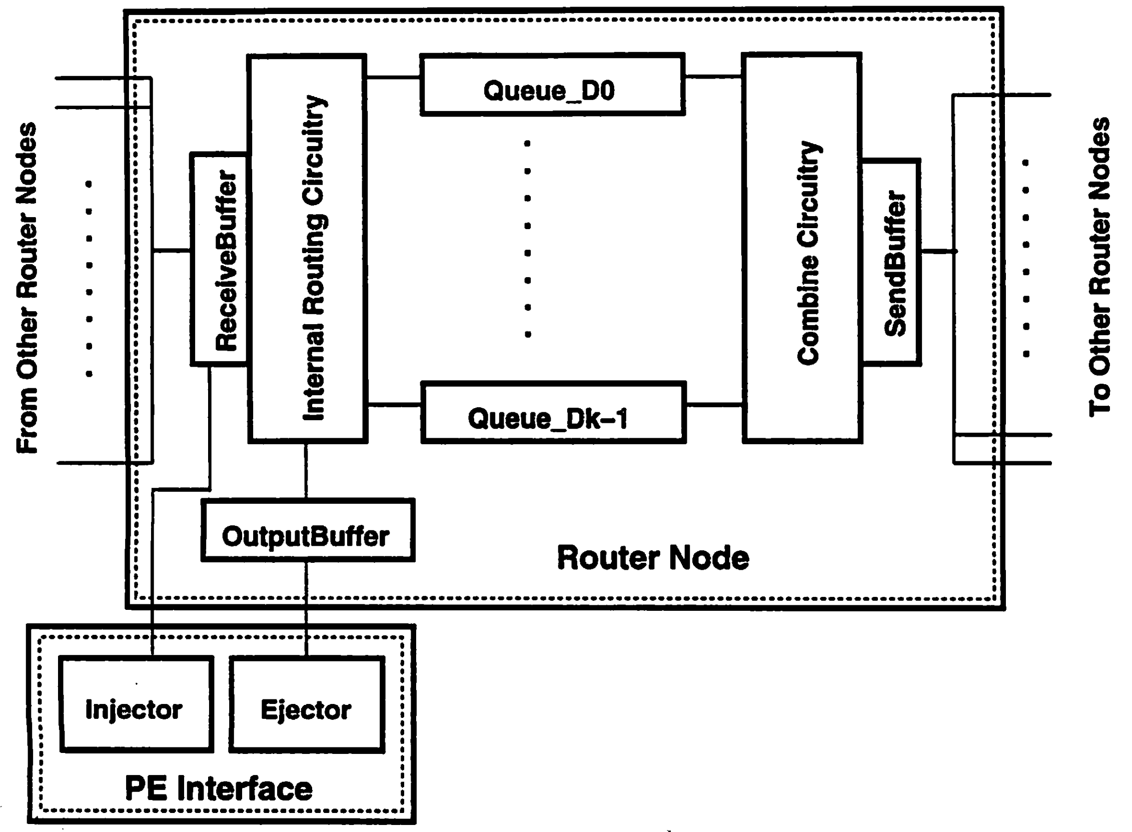


Figure 10.8. Shown is the ENPASSANT packet-switched network switch model.

10.3.3.2 The Packet-Switched Network Simulator

Besides topology and the number of connections between routing nodes, the primary architectural decisions have to do with switch design. We now describe the ENPASSANT PIN switch model used in the packet switched network simulator. See Figure 10.8.

A packet is input into the ReceiveBuffer either from the injector (PE interface) or from other routing nodes. From the ReceiveBuffer, the packet is routed to the appropriate queue, or the OutputBuffer. The combine circuitry combines packets having the same destination and moves the result into the SendBuffer. The OutputBuffer is read by the ejecter (PE interface). The routing switch has a number of queues equal to the number of connections. The queues themselves have a fixed, but specifiable, number of slots.

The routing algorithm that is implemented is based on the one used by the CM2 router network. This is part of the CM2 algorithm as described by Almasi and Gottlieb [7]:

> It consists of a number of 12 steps, or "dimension cycles"; during step i the message is sent to the adjacent node in dimension i if the ith bit of its router address is 1. ... Each time [the packet] is sent along some dimension, the corresponding 1 is set to 0. When the address is all 0s, the message has arrived.... The set of 12 dimension cycles

is sometimes called a "petit cycle" .... Since the algorithm accesses the dimensions in a cyclical fasion, a message that runs into a conflict with another message during a dimension cycle must wait at least one full petit cycle before it gets a second chance to move along that dimension.

The following routing algorithm is implemented by our PIN simulator.

```
DO UNTIL all packets have been routed
    DO $i$ from 0 to $k-1$ (During every dimension cycle)
        If there is a packet in the OutputBuffer, then the Ejector removes it.
        Combine packet in head of Queue_$i$ with any other packet in
            Queue_$i$ that has the same destination and move into SendBuffer.
        Send packet in SendBuffer to the ReceiveBuffer in the next node in
            dimension $i$.
        If no packet was received, then the next packet in the InjectorQueue
            is moved into the ReceiveBuffer.
        New packet in ReceiveBuffer gets routed (internally) to the appropriate
            queue. Depending on the nodes that still need to be traversed, this
            is either the current dimension, the $(i+1)mod(k-1)$st dimension,
            or the DoneBuffer. If correct destination queue is filled, then the
            packet is mis-routed.
```

The parameters that can be varied in the ENPASSANT packet switched network simulator are the number of router nodes, the topology (number of dimensions), the queue sizes within the nodes, the dimension cycle latency, and whether the links between nodes are uni-directional or bi-directional.

Although the packet switched network simulator is quite flexible, it obviously is not a parameterized model of all possible routing configurations. It does have a number of advantages, however.

- It can be used to approximate the CM2 global communication network.
- It models likely variations of the CM2 network: changes in queue size, changes in topology, and changes in dimension cycle latency.
- Because of the comparatively fine granularity of SIMD PEs, the router nodes are unlikely to be significantly more complex than the ones the simulator models. In particular, as long as the network subsumes the petit cycle/dimension cycle model, this simulator is likely to be sufficient.

See Figure 10.9 for a sample result. A random permutation for an array of 64K PEs was simulated on a 1024 node (10 dimensional) and on a 2048 node (11 dimensional) hypercube networks. The fraction of PEs sending packets was varied and the effect of that variation measured with respect to the number of dimension cycles the route took to complete. For both networks, a linear fit was found to be highly statistically significant. In this respect, the PIN simulator matches the CM2 router network which has the property that for large numbers of packets, the routing time is proportional to the number of packets being routed [27].

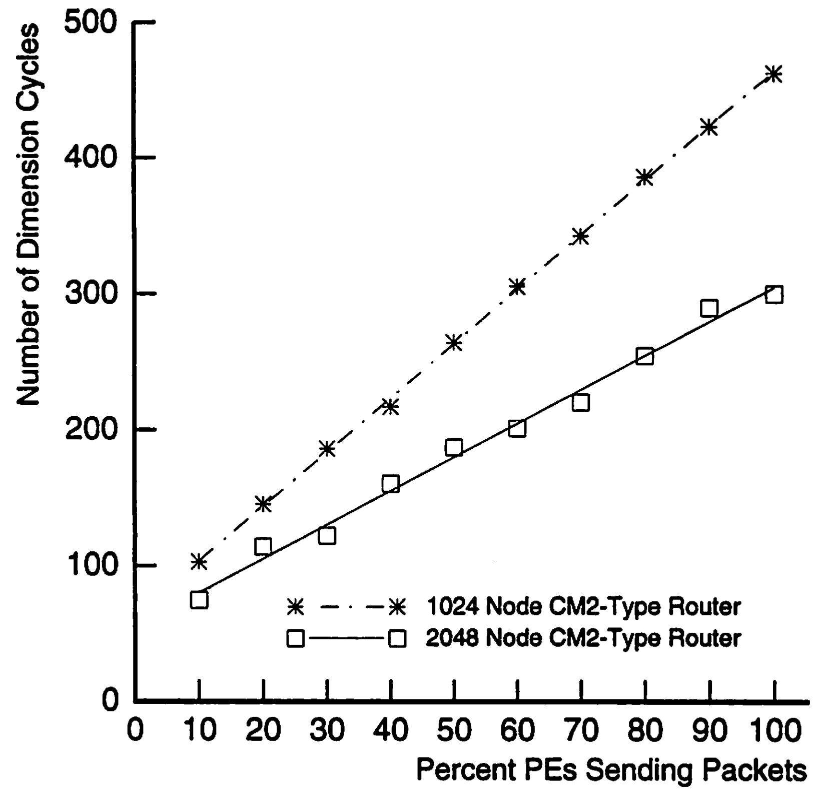


Figure 10.9. Number of dimension cycles required to route random patterns of various densities as obtained by simulating CM2-like routing networks. Size of simulated array is 64K PEs.

10.3.3.3 A Packet-Switched Network Case Study

We close this subsection with a case study wherein we examine the advantages, if any, of increasing the number of dimensions in a k-ary n-cube packet switched network. This question has received attention in both the performance evaluation and the theory communities (see e.g. [49]) because of its importance in determining the topology of the networks to be built for the next generation of massively parallel processors. These studies have usually assumed randomly distributed and independent routing destinations, packet generation by Poisson process, etc. We instead run experiments on communication patterns arising from real applications.

The routing patterns are derived from the reduction of non-uniform regions, taken from a region segmentation algorithm run on images of outdoor scenes. In other words, for all regions, all PEs in a region send a message to an arbitrarily selected 'leader' PE. See Figure 10.10 for an example of such a segmented image. The pattern is interesting because of its importance, and also because of the difficulty in modeling it analytically. Regions can vary widely in shape and size. There is a great deal of congestion, not just because of the many-to-one nature of the pattern, but also because of the fact that the leader PEs can be proximate. The latter condition means that many colliding packets cannot be combined, unlike the situation in regular pattern reductions.

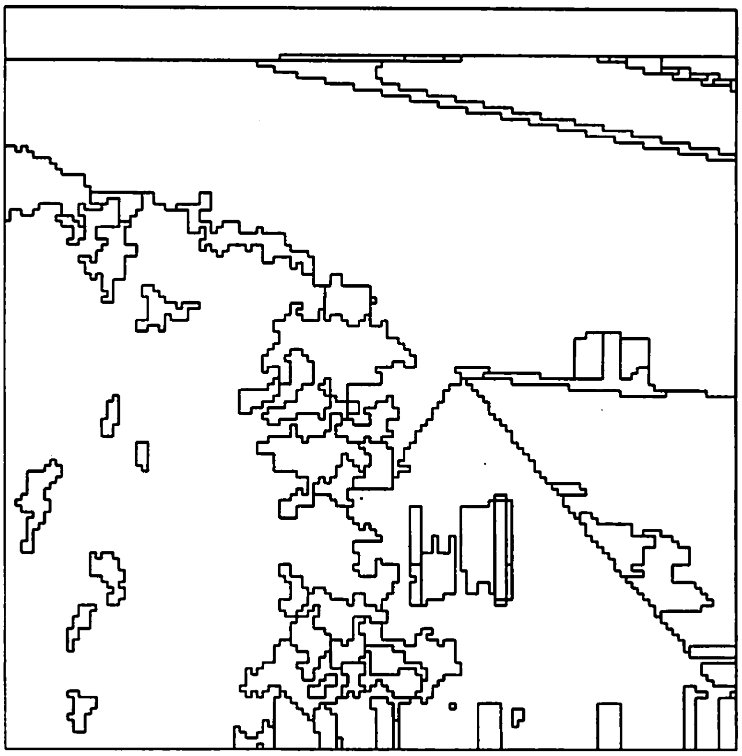


Figure 10.10. A partially segmented house scene image.

In the case study, the variation of two network parameters is measured with respect to the number of dimension cycles the routes take to complete on patterns of the type just described. The parameters are the number of dimensions and whether the network supports bidirectional or only unidirectional transfers. By counting dimension cycles, we are effectively holding the bandwidth among the nodes constant, a condition that favors the higher dimensional networks. This is because it is much easier to lay out wires in 2 or 3 dimensions than in 6 or 12. For each set of parameters, trials were run on five images of the type shown in Figure 10.10.

The results shown in Figure 10.11 are of a network with 4K router nodes, an array with 16K PEs, and a virtualization factor of 1. The error bars represent 95% confidence intervals. For unidirectional networks, the performance gain tails off rapidly after 3 dimensions and definitely after 4. The 6 and 12 dimensional networks show no benefit at all. Because of the bias towards higher dimensions, this indicates that 2, or at most 3, dimensional unidirectional networks are recommended. The result for bidirectional networks are similar: there is a performance gain when the number of dimensions goes from 2 to 3; after that the performance actually degrades. However, the most significant result from this case study might be the comparison between the performance of the 2 dimensional unidirectional and bidirectional networks: The performance of the latter is *twice* that of the former. This implies that giving a 2 dimensional network the capability of routing both up and down dimensions—on alternating dimension cycles so that the wires can be reused—is likely to be important.

One final note on this experiment: for 6 dimensional networks the performance for a bidirectional network is actually worse than that of a unidirectional network. To see the reason for this, we compute the expected number of dimension cycles required to route a packet with no congestion. For a k dimensional network with n nodes per dimension and $n > 2$, the expected number of dimension cycles required to traverse a unidirectional network is $\bar{d}_u = k \times (n-1)/2$; for bidirectional networks it is $\bar{d}_b = k \times n/2$. This disparity is significant for the 6 dimensional network because when $n = 4$, $\bar{d}_u = 1.5$ while $\bar{d}_b = 2$. This difference is too great to be compensated for by the lower congestion of the bidirectional network.

10.4 Implementation and Performance of MVM Communication

As noted at the beginning of this chapter, we are interested in evaluating the effect of interPE communication network design on the performance of MVM communication instructions. ENPASSANT supports four communication network models: mesh, mesh plus a broadcast network, mesh plus a circuit switched network, and mesh plus a packet switched network. The MVM instructions with which we are concerned are as follows. As always in this study, each physical PE can contain multiple data elements of the input and output Planes.

- Route. This operation takes four parameters. Data elements in the input Plane are reordered and moved into the output Plane according to the contents of the row and column index Planes. If multiple input data have the same destination, only one is retained.

- OpRoute. This operation is the same as Route except that multiple input data having the same destination are combined according to the operator specified, typically + or MAX.

- OpSegScan. OpScan is defined as follows: Given a set $[x_1, \ldots, x_i]$ of n elements with each element assigned to a different processor and a binary associative operator $*$ (the Op, compute the n S_i's, where $S_i = x_1 * x_2 * \ldots x_i$, leaving the ith prefix sum in the ith processor. A *segmented* OpScan (an OpSegScan) consists of a sequence of OpScan operations that are applied to disjoint sets of data. The partitions of the input data set are specified by a binary input with the same number of elements as the input data set; the binary input can be viewed as a set of *barriers*. At each location in the barrier set where a 1 appears, a new

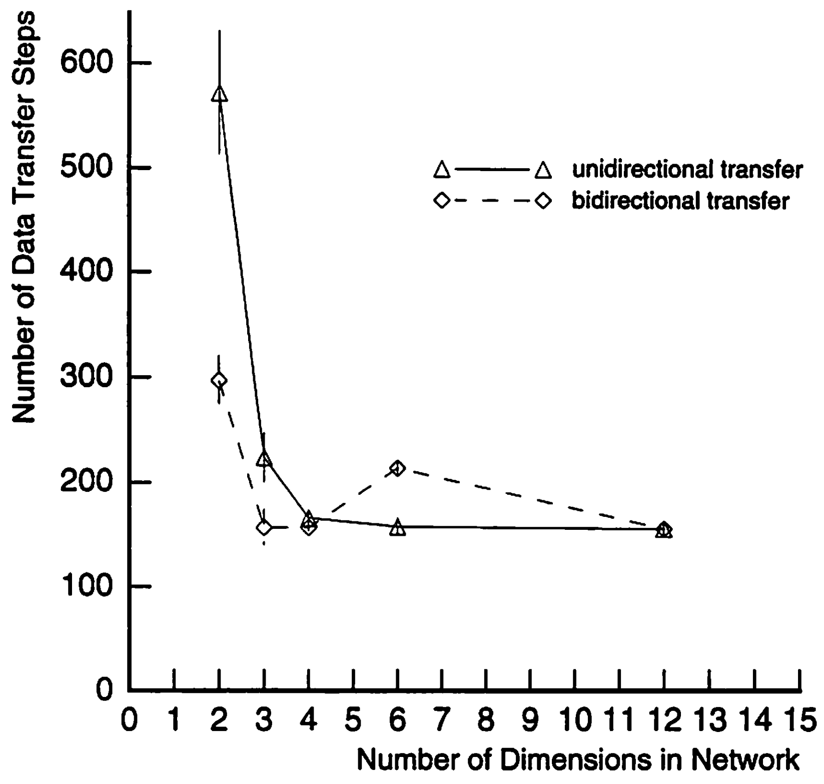


Figure 10.11. Effect of varying the number of dimensions on the number of dimension cycles required to reduce the regions shown in images of the type shown in Figure 10.10.

OpScan computation is initiated. Segmented scans take only slightly longer than regular scans; see [92] for details. ICL supports three types of OpSegScan operations: within Planes, within all Rows in a Plane in parallel, and within all Columns in a Plane in parallel.

- RegionBroadcast. This operation takes four parameters: the input and output Planes, a Plane specifying the region boundaries, and a Plane specifying the senders. Data elements in the input Plane that correspond to senders are broadcast to all elements of the output Plane within the same region.

Table 10.1 displays these network/instruction combinations and indicates which are supported directly in hardware and which require emulation.

In the following subsections, the effect of the target machine architectural specification on the performance of the four MPA communication instructions is examined. Each subsection also sketches how those instructions are implemented for the various communication network models. The emphasis here is on the mesh and the mesh plus broadcast models. This is because they are involved in more emulations (the instruction implementations involving them are non-trivial) than the other two models, and because the performance of those emulations depends on architectural

	MPA Communication Instruction			
Communication Network	Permutation Route	RegRoute (non-uniform reduction)	(Segmented) Scan	Region Broadcast
Mesh	emulation needed	emulation needed	emulation needed	emulation needed
Mesh + Broadcast	emulation needed	emulation needed	emulation needed	supported directly
Mesh + Circuit Switched	supported directly	emulation needed	emulation needed	emulation needed
Mesh + Packet Switched	supported directly	supported directly	supported directly	emulation needed

Table 10.1. This table shows which communication instruction/network combinations are supported directly and which require emulation.

features that are more central to this dissertation: the size of array, nearest neighbor bandwidth, and PE datapath widths.

10.4.1 Scans

ICL supports both segmented and non-segmented scan instructions for rows, columns, and Planes. A number of combining operators are allowed. These Scan instructions are implemented directly on the packet switched networks; emulation must be used for them to run on the other networks.

10.4.1.1 Scan Implementations

The circuit switched network is not effective in executing or emulating this operation. The Scan emulation on a mesh follows almost immediately from the definition.

One method of emulating Scans on broadcast networks is to use the well known algorithm for executing scans on lines and rectangles of processors. See, for example [72, 70]. For small array sizes, however, these algorithms have little advantage over simply using the mesh emulation.

10.4.1.2 Scan Performance

The performance of the Scan operations depends only on the architectural parameters and the virtualization factor. Scan therefore does not need to be simulated by the packet and circuit switched routing networks, but rather, the performance can be input directly as a parameter. Also, for the array sizes we are considering, using broadcast for the Scan instruction implementation has little benefit, so those results are omitted as well.

We now present the effect on the Scan instruction on mesh networks of varying the array size, virtualization factor, ALU/datapath width, and nearest neighbor transfer latency. The specific instruction examined is a segmented row scan with the plus operator. Figure 10.12a shows the effect on performance (in target machine cycles) of varying the ALU/datapath width and the array size/virtualization factor. As we said above, Scan operation performance is not data dependent. Note that the performance improvement due to increasing the ALU/datapath widths levels off rapidly after ALU/datapath width = 2. This is due to the fact that at this point the nearest neighbor transfers dominate. This can be seen in Figure 10.12b which shows the effect on performance of increasing nearest-neighbor path width along with the ALU/datapath width: significant performance improvement continues through path widths = 32.

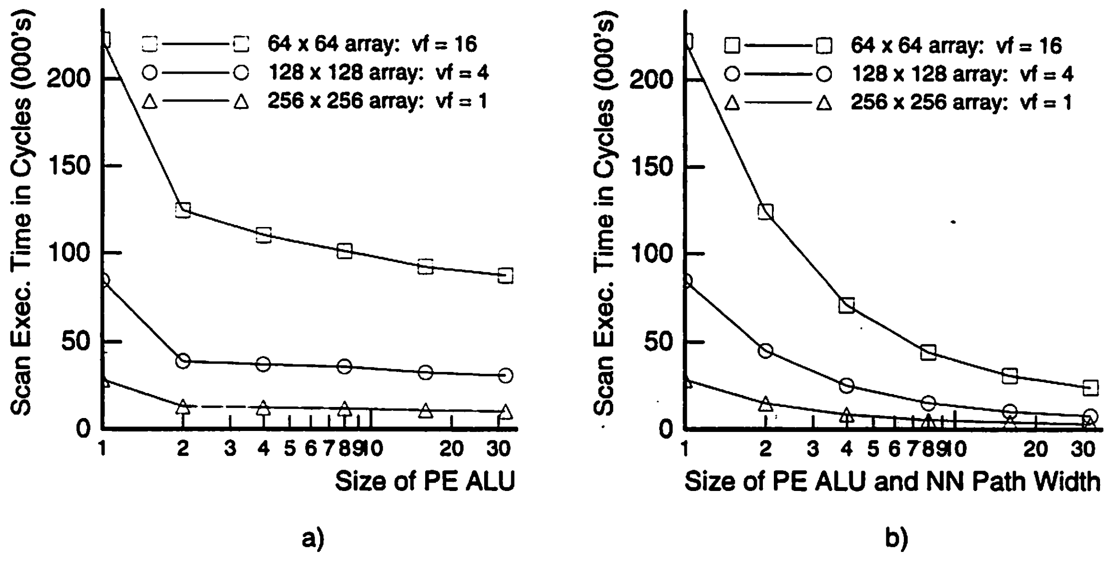


Figure 10.12. Effect on +Scan performance on the mesh of varying: a) the ALU datapath width and the number of PEs, and b) also varying the mesh path width.

10.4.2 Route

The Route operation is implemented directly on the circuit switched and packet switched networks as was shown in Sections 10.3.2 and 10.3.3. However, mesh and broadcast networks do not support Route directly and so must use emulation. These emulations and how their performance depends on the underlying architecture are now discussed.

10.4.2.1 Route Implementations

The method used for both networks is the greedy algorithm described in [70]. In its simplest version, packets are first routed along rows (X-channels) until the correct column is reached and then up the columns (Y-channels) to the destination. When the virtualization factor v is greater than 1, then the method is modified as follows. Instead of having only an X- and a Y-address, each packet also has a W-address. The W-address indicates the destination ***tile***. The tile indicates which of the v virtual processing elements being emulated by the physical processor is the true destination.

Again a greedy strategy is implemented. A packet originates in the W-channels and is routed until the correct (ith) tile is reached (in W-channel[i], still within the PE). The packet is transferred to the X_i-channel and routed along the rows until the correct column is reached. The packet is then transferred to the Y_i-channel and routed along the columns to the destination. The advantage of this method is that queues are required neither at the input nor at the output. Pseudo-code for a simplified version of the algorithm follows.

```
1. For Tiles 0 to $v - 1$
            If packet in the $Y_i$-channel has reached its destination,
              then move to $Output_i$
2. For Tiles 0 to $v - 1$
            Route packet in $Y_i$-channel to next PE
3. For Tiles 0 to $v - 1$
            If packet in $X_i$-channel has reached its destination,
              and $Y_i$-channel is clear, then move to $Y_i$-channel
            Else mark as blocked
4. For Tiles 0 to $v - 1$
            If $X_i$-channel packet is not blocked, and destination is clear,
              then route packet in $X_i$-channel to next PE
5. For Tiles 0 to $v - 1$
            If packet in $W$-channel[i] has reached its destination,
              and $X_i$-channel is clear, then move to $X_i$-channel
            Else mark as blocked
6. For Tiles 0 to $v - 1$
            If $W$-channel[i] packet is not blocked, and destination is clear,
              then route packet in $W$-channel[i] to next PE
```

10.4.2.2 Route Performance

The performance of the Route instruction on the four communication models is as follows.

- For circuit and packet switched communication networks, the performance of the MVM Route instruction is nearly equivalent to the latency of the network; in other words, there is very little additional overhead. We have shown how circuit and packet switched router networks perform under random permutations of various densities (see Figures 10.7 and 10.9).

- Mesh network performance is examined in this section.

- The performance of the mesh plus broadcast model is similar to that of mesh networks for dense permutations, but is up to a factor of 20 faster for sparse permutations. See [70] for details.

The Route instruction on mesh networks depends on the array size, virtualization factor, datapath width, and nearest neighbor transfer latency. The pattern being routed is a random permutation; we show in [70] that the variance is small among this class of routing patterns.

Figure 10.13a demonstrates how increasing the ALU and datapath widths, as with the +Scan instruction, improves performance only up to a point. But this time, although the nearest neighbor connection bandwidth is important, its effect is not nearly as much as with the +Scan instruction. Notice in Figure 10.13b that increasing the nearest neighbor connection bandwidth along with that of the ALU and datapath results in less than a factor of 2 speed up.

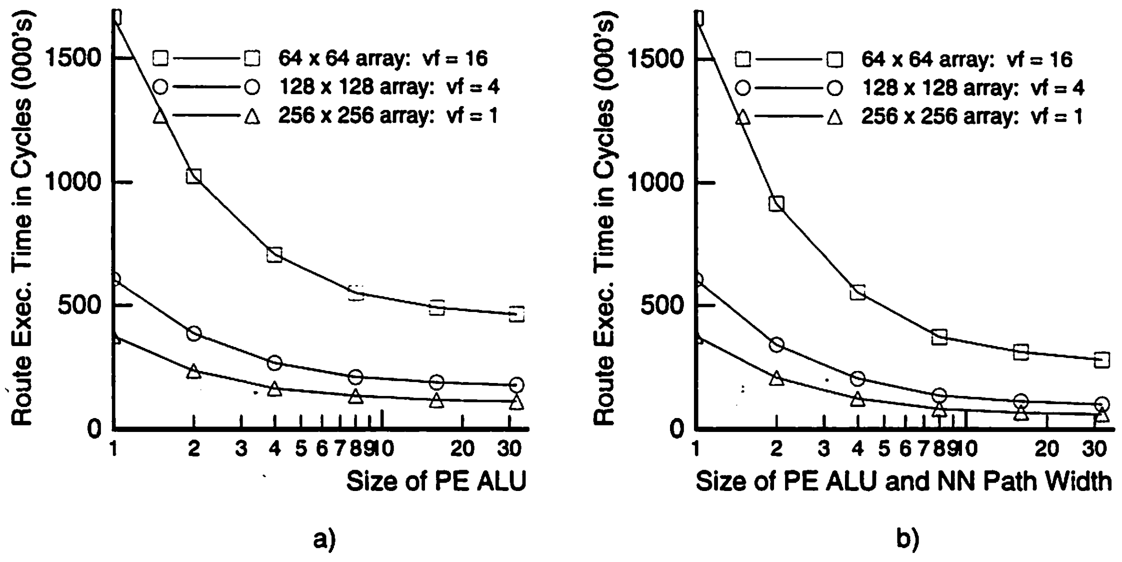


Figure 10.13. Effect on Route performance on the mesh of varying: a) the ALU datapath width and the number of PEs, and b) also varying the mesh path width.

Examining the Route instruction emulation code in subsection 10.4.2 reveals why: unlike the +Scan emulation, the +Scan emulation is data dependent. Therefore a number of control operations (such as compares) internal to the PE must be executed during every iteration.

The performance of non-random permutations can be up to a factor of 2 slower than that of random permutations, but is very rarely worse [70]. The case of many-to-one routing is examined in the next subsection.

10.4.3 OpRoute: Routing with Combining

In this subsection we examine the subset of reduction instructions (OpRoutes or route with combine) that operate on non-uniform regions. Uniform reductions require execution time similar to OpScans.

10.4.3.1 Region OpRoute Implementations

As with Route instructions, OpRoutes are implemented directly on the packet switched networks. Mesh and broadcast networks, however, do not have that capability and so must use emulation. In the latter case, the algorithm shown in Section 10.4.2.1 is easily modified to support combining. See [70] for details.

Circuit switched networks, however, are also not capable of implementing the operation directly. There are two possible methods of emulating OpRoute: 1) using multiple passes through a circuit switched network (combining packets at the destination), or 2) using the mesh network emulation. Since the first alternative could require hundreds of thousands of passes, the second alternative must be used.

10.4.3.2 Region OpRoute Performance

The performance of the OpRoute instruction on mesh networks again depends on the array size, virtualization factor, ALU/datapath width, and nearest neighbor transfer latency. The specific instruction examined is +Route; the routing pattern is a reduction of the regions shown in Figure 10.10.

The graphs in Figure 10.14a show the effect of increasing the ALU/datapath width while the graphs in Figure 10.14b show the effect of increasing the nearest neighbor path width as well. Note the difference between Figure 10.14a and Figure 10.13a: because arithmetic operations are executed during every iteration of the reduction algorithm (for combining), the performance is more dependent on the ALU width (past width of 2) than Route without combining.

The behavior on packet switched networks of reduction routing in non-uniform regions was shown in Figure 10.9.

10.4.4 Region Broadcast

RegionBroadcast is implemented directly on the broadcast networks; it must be emulated on the mesh and packet switched networks. And again, the circuit switched network is not effective in executing or emulating this operation.

10.4.4.1 RegionBroadcast Implementations

The most efficient implementation of RegionBroadcast on a mesh is usually the brush-fire method. That is, all virtual PEs broadcasting data send a copy to the virtual PEs in the NEWS directions that are still in the region. The receiving PEs combine the information from each direction using the OR operator. The process is repeated until no values change in consecutive iterations.

RegionBroadcast is implemented on packet switched networks through the repeated use of segmented scan operations. See [98] for details.

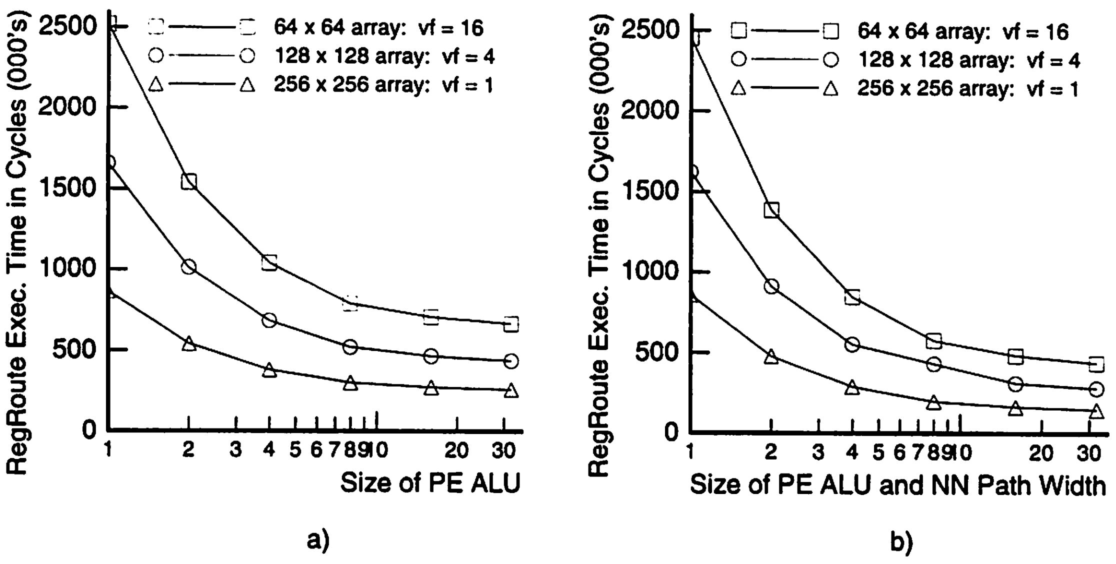


Figure 10.14. Effect on +Route performance on the mesh of varying: a) the ALU datapath width and the number of PEs, and b) also varying the mesh path width.

10.4.4.2 RegionBroadcast Performance

For broadcast networks, the RegionBroadcast instruction performance is proportional to the propagation delay through the network. On a 256 x 256 CAAPP, this delay is 10 cycles. See [24] for a discussion of delay issues.

The performance of RegionBroadcast on meshes has very similar dependencies as the performance of OpRoute on non-uniform regions. See Figure 10.15.

On a packet switched network, typically 60 to 70 segmented scans are required to implement a region broadcast in images such as the one shown in Figure 10.10.

10.5 Chapter Summary

In this chapter we have presented the communication analyzer component of ENPASSANT and some examples of how it is used. We have built detailed simulators of two classes of networks: packet switched k-ary n-cube networks, and circuit switched multi-stage interconnection networks. Although there have been innumerable studies of these classes of networks, relatively few have been in the context of MPAs and fewer have shown the performance effects of design changes with respect to communication patterns arising during real program executions. One of the results obtained here is that in k-ary n-cube packet switched networks, higher dimensionality does not seem to provide a significant benefit for an important group of patterns used in machine vision. Another result is that for this set of patterns, adding bi-directional transfer capability doubles the performance of the two dimensional version of the nework. Other results in this chapter deal with the effects of PE and nearest-neighbor datapath sizes, as well as array size, on the performance of communication instruction emulations.

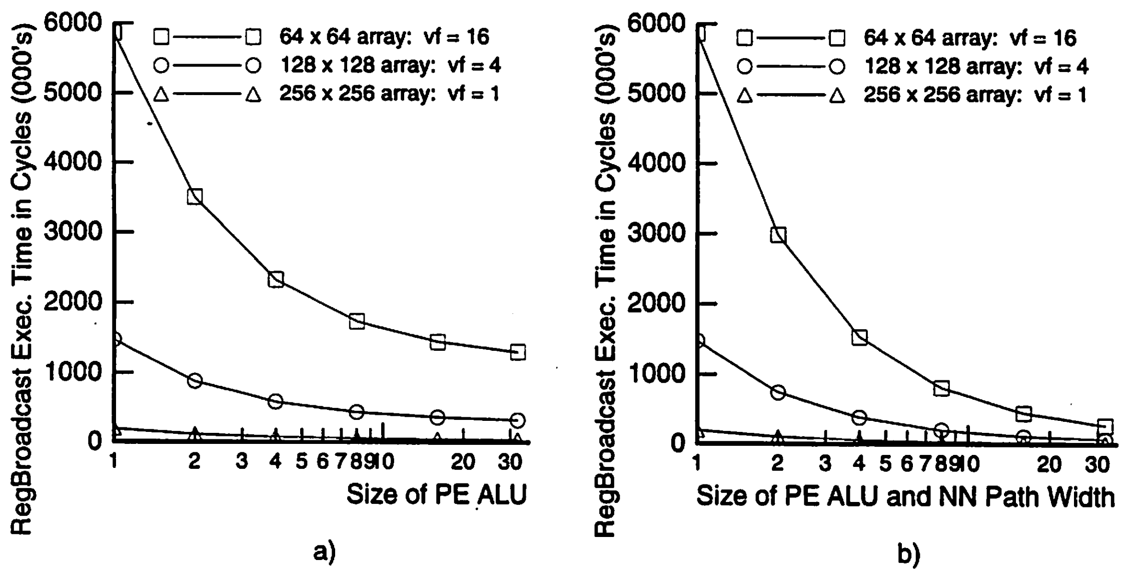


Figure 10.15. Effect on RegionBroadcast performance on the mesh of varying: a) the ALU datapath width and the number of PEs, and b) also varying the mesh path width.

PART V

RESULTS AND FUTURE DIRECTIONS

Chapter 11

Case Study in MPA Evaluation

11.1 Goals of MPA Evaluation

An ideal architectural development environment would specify the precise architecture that gives the best performance for a particular workload under the constraint of various cost factors such as price, power consumption, size, reliability, etc. Since ENPASSANT does not account for cost, we rely on the user to know what the effects on cost of a particular architectural specification and to constrain the designs to feasible alternatives. As we have already seen, however, these constraints do not prevent us from obtaining important information about MPA architectures.

In particular, without specific cost knowledge, we are looking for the following types of results:

- Sudden changes in performance with respect to a small change in a parameter. These can be found by examining graphs for non-linearities. These non-linearities can sometimes be predicted from analogous experiences with other hardware (working sets); sometimes they are inherent in the nature of MPAs and their data (routing locality).

- Which components improve performance enough to justify their expense and which do not.

- Balance, or alternatively, bottlenecks. For example, if 90% of the execution time of all the test programs is spent on memory access, that is clearly a bottleneck. But if secondary storage is never accessed, then the primary storage is probably larger than necessary. In particular, we seek balance among the three primary components—memory, datapath, and communication—and also within those components, where applicable.

- Break-even points in trade-offs. It is sometimes possible, for example, to trade off speed of a particular component (number of cycles required to perform an operation) for overall clock speed. Finding the break-even points in such trade-offs is essential.

In the next section we examine the performance of the test suite programs with respect to variations in array size (level of PE virtualization) and memory hierarchy.

11.2 How Array Size Affects Memory Hierarchy Performance

As we have described previously, the common method for handling the situation where there are too many elements in a parallel variable to map one-to-one to the PEs of an MPA array is to map the elements to the required number of virtual PEs. The physical PEs then emulate the virtual PEs. The effect of reducing the PE array size is that each physical PE must emulate a larger number of virtual PEs. Put another way, the virtualization factor (VF) is increased.

In this section, we examine the interaction of array size (the inverse of the VF) and memory hierarchy design. We begin by examining in detail the effect of varying the array size with respect to the ARPA IU Benchmark and showing that MPA performance is greatly enhanced by adding another level to the memory hierarchy. We then examine a function with a particularly large working set—mesh Route emulation (see Section 10.4.2.1)—and show similar conclusions. After that, we present some data from the rest of the test suite. Since simulating codes with a VF of 16 or greater is very expensive in disk space, these last results were only generated for VFs of 1 and 4.

We use a CAAPP-like datapath model in generating these results; however, the same basic conclusions hold for all datapath models.

11.2.1 The ARPA IU Benchmark II

The graph in Figure 11.1 shows the effect of changing the array size on the execution time. The program run was the IU benchmark; a register file with size equal to 100 bytes was used. The 'ideal behavior' shown was derived by multiplying the execution time with the $VF = 1$ by each successive VF. We refer to this behavior as ideal because it indicates no slowdown due to overhead in virtual PE emulation.

There are two things to notice. The first is the nearly linear slowdown (over the ideal behavior) that occurs when the VF is 2, 4, or 8. This can be attributed to the ever greater proportion of time needed to emulate communication among virtual PEs. The second is the leap that occurs when the VF goes to 16. The reason for this can be seen in Figure 11.2: for smaller VFs, most of the working set can fit into a register file of 100 bytes per PE; this is no longer the case when the virtualization factor reaches 16.

Figure 11.2 shows the effect of register file size on the percentage of the total execution time spent on memory access for the VFs shown in Figure 11.1. The practical result given in Figure 11.2 is the minimum register file size for different VFs so that memory fetches do not a dominate the total execution time. The memory access time is assumed to be 5 times that of an arithmetic operation. For virtualization factors of 1 and 2, only a relatively small register file (25-30 bytes per PE) is needed. The small size is not surprising because most of the Plane types used by the benchmark require only single byte storage. It is also apparent, however, that the working set is roughly proportional to the VF. This means that as the PEs cycles through the emulation of VF virtual PEs, running the code segments in round-robin fashion, very little of the ith context

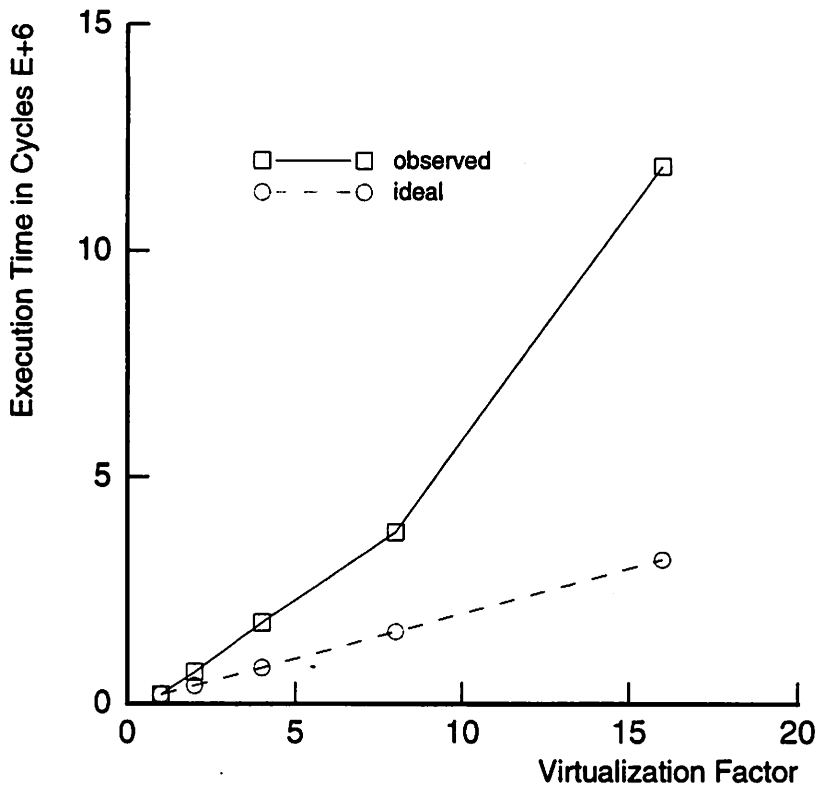


Figure 11.1. Effect of array size on the execution time of the IU benchmark.

remains when emulation on the *i*th virtual PE begins again. Or put another way, the working set of virtual PE *i* will have been swapped out of the register file before it is time for virtual PE *i* to be emulated again. This is not surprising since for much of the code there is no overlap between the individual virtual PE working sets. In any case, the result is that the register file size must increase proportionally to the VF in order to insure that memory access does not dominate the computation.

Since the register file size cannot be increased indefinitely, however, the architectural answer is to add another level to the memory hierarchy. Figure 11.3 contains the hit rates of caches of various sizes on the memory reference traces generated for Figure 11.2. For simplicity, the cache is assumed to be fully associative with a block size of 8 bytes. As expected, the hit rate improves with the size of the cache. Also expected is the result that a smaller cache suffices for a smaller virtualization. Less obvious is that the hit rate should *decrease* as the register file size increases. This is explained as follows: the larger register file size reduces the absolute number of memory references, thereby decreasing the locality of those that remain.

Another result from Figure 11.3 is that a cache size of 400 bytes per PE suffices to achieve a 90% hit rate even for a virtualization factor of 16.

In Figure 11.4 we show the effect of increasing the VF on the performance of the benchmark program for a model with 400 bytes cache per PE, but with a register file that has been reduced

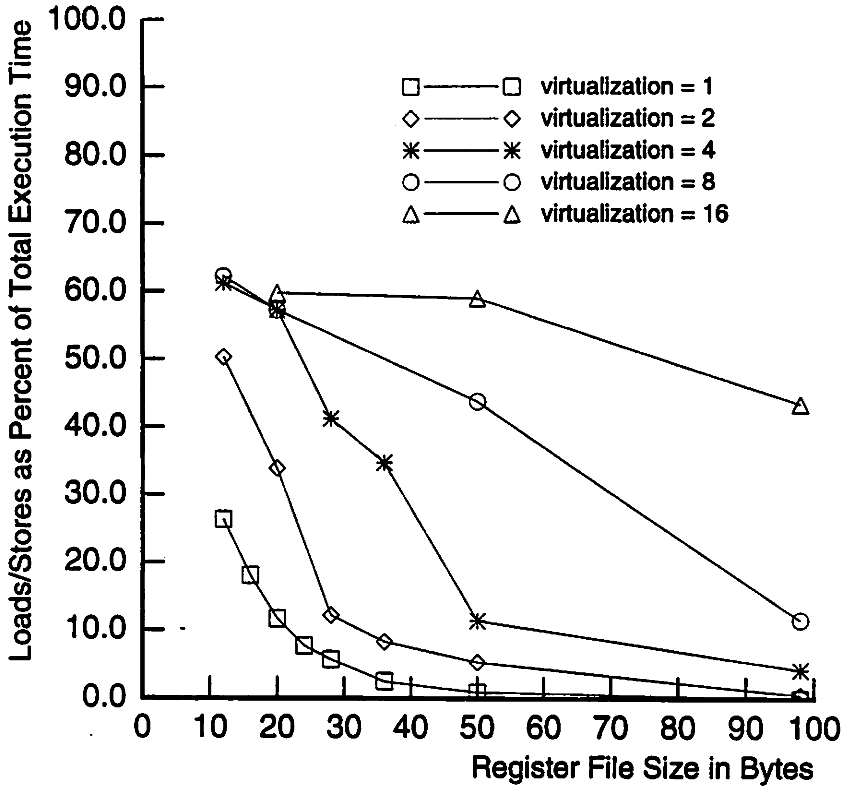


Figure 11.2. Effect of the register file size on the percentage of the total execution time spent on memory access for different virtualization factors.

from 100 to 50 bytes. Note the improved performance for VF = 16 with no loss in performance for smaller VFs.

11.2.2 Route Emulation on Meshes

We now look at the effect of array size on memory performance with respect to a function that has the largest working set we have yet encountered: the Route emulation on a mesh processor (see Section 10.4.2.1). Figure 11.5a shows that the working sets for virtualization factors of 1, 4, and 16 are approximately 50, 200, and 800, respectively. Somewhat surprisingly, however, the cache requirement for this function is substantially smaller than that needed to run the ARPA IU benchmark efficiently: Figure 11.5b shows that the knee of the curve for a VF of 16 is at a cache size of 200-250 bytes per PE, rather than the 350-400 bytes needed previously.

The reason for this decrease in the cache size requirement has to do with the nature of the function being examined: the communication emulation does not force the processor to cycle through virtual PE code segments as the IU benchmark code does. Rather, it is a monolithic algorithm that happens to have VF dependencies. In particular, the Route emulation uses several arrays of Planes having a number of elements equal to the VF. Because of this difference, the

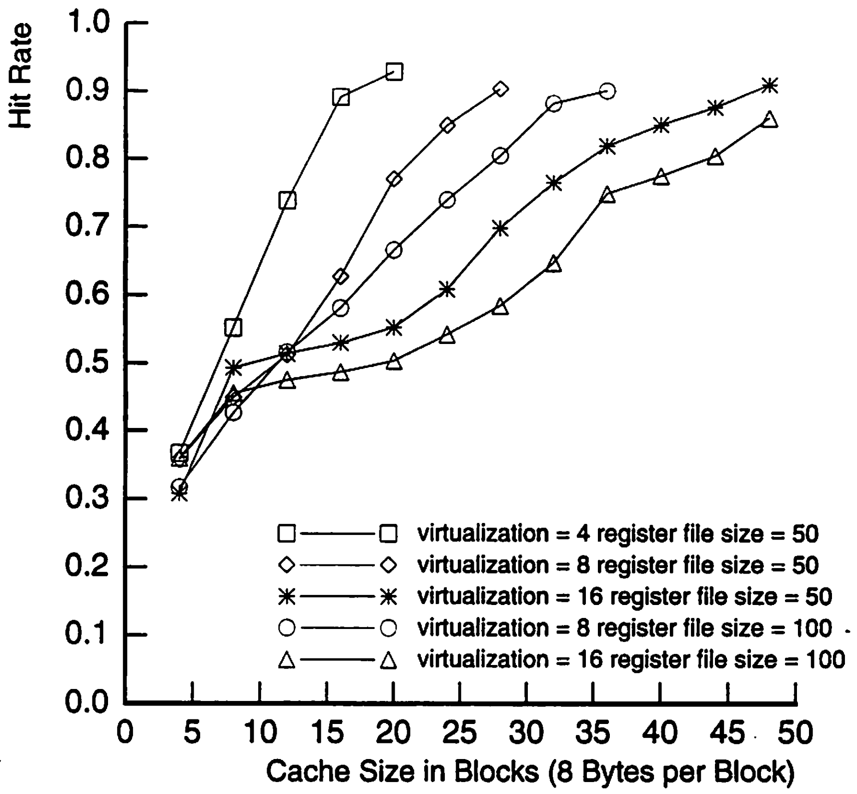


Figure 11.3. Plot of the hit rate versus the cache size in 8 byte blocks. The cache is fully associative.

memory reference locality properties of the mesh Route emulation are more similar to the locality properties of serial processor codes than is typical in the other MPA programs. In other words, the familiar rule of spatial locality of memory references (if one array element was recently accessed, another one is likely to be soon after) holds more for the Route emulation than it does for the IU benchmark.

11.2.3 The Test Suite

In Figure 11.6 we show the effect of the VF on the fraction of the total execution time spent on memory references. We find that for the programs with significant working sets (all but the curve-fitting filter), the register set must grow roughly proportionally with the VF in order to maintain performance. This agrees with the result shown in Figure 11.2.

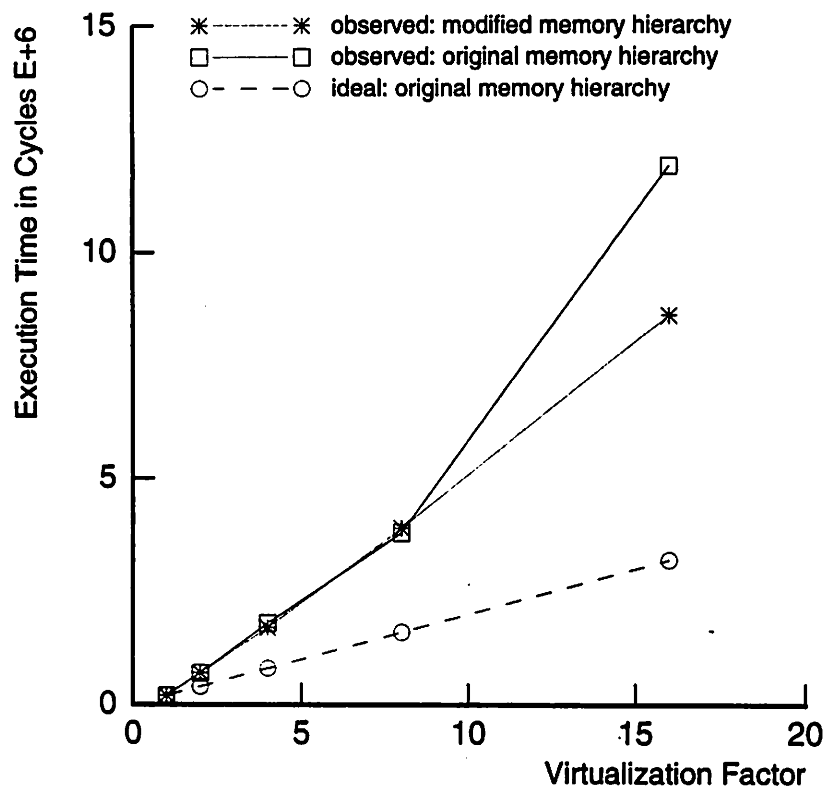


Figure 11.4. Effect of PE virtualization factor on the execution time of the IU benchmark as a function. The new memory hierarchy has a cache, but smaller register file size.

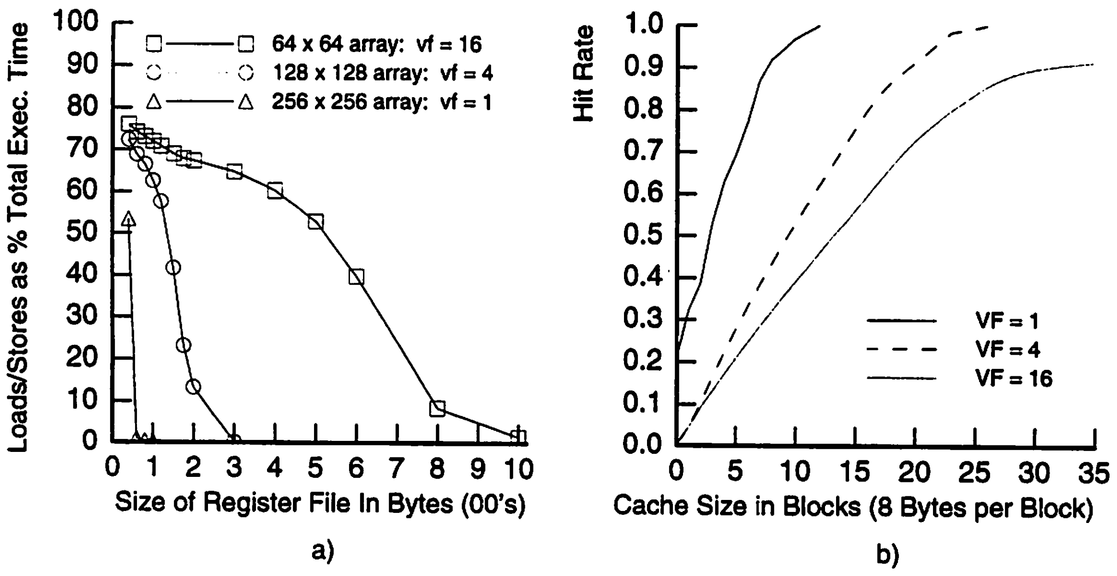


Figure 11.5. For the mesh emulation of the Route instruction: a) effect of the register file size on the fraction of time spent on memory references, and b) effect on the hit rate of the cache size in 8 byte blocks for a fully associative cache and a register file size of 60 bytes.

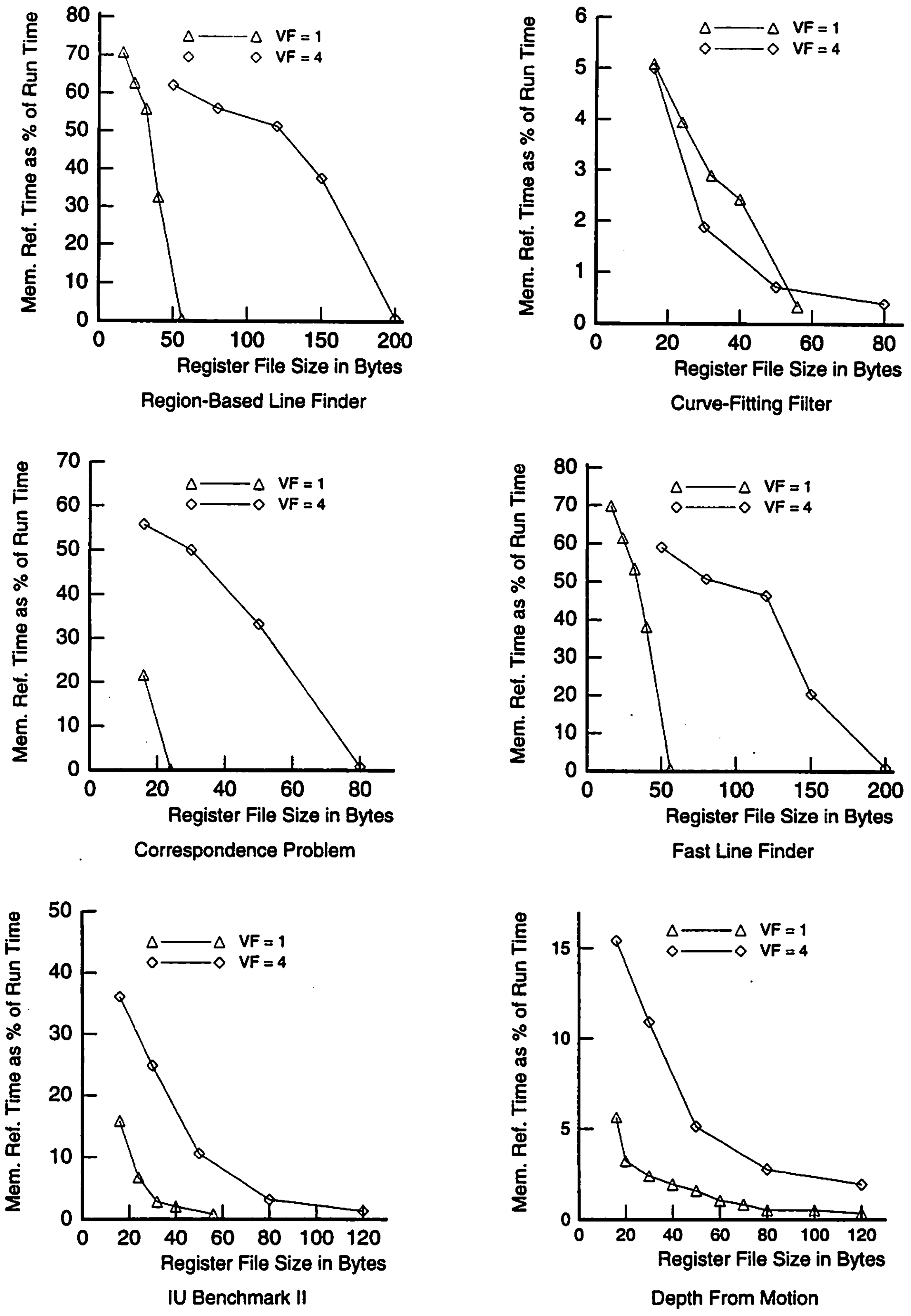


Figure 11.6. Shown is the effect of changing the PE virtualization factor from 1 to 4 (reducing the array size by a factor of 4) on the fraction of the execution time spent on memory references when running several of the test suite programs.

Chapter 12

Conclusion

12.1 Summary

The goal of this dissertation is to facilitate the application of empirical techniques to the study of massively parallel array architectures. This has been accomplished by constructing a simulation system that performs the following functions:

- inputs spatially mapped application codes written in a data parallel class library extension to C++;
- inputs architectural specifications such as the size of the array (number of PEs), type of routing network, and many others; and
- outputs the number of cycles required for total program execution as well as detailed profile information.

The characteristics of the simulation system are:

- **Flexibility.** ENPASSANT allows the user to specify a large number of features and parameters from the MPA architectural design space. Some of these are available on current MPA designs, many others are speculative. Programs written for ENPASSANT can be used to evaluate any target architecture without recoding.
- **Accuracy.** We have shown that for any particular architectural model, the results obtained will closely resemble the output from a detailed simulation based on that same model.
- **Performance.** The results of a particular evaluation are obtained from one to several orders of magnitude faster than when using detailed simulation, depending on the parameter or feature being varied.
- **Fairness.** We have used selective recoding to insure that the application codes do not skew results unfairly due to choice of algorithm.
- **Portability.** ENPASSANT requires only a standard environment (C++, UNIX, X-Windows) and a workstation with a good sized memory and disk drive.

ENPASSANT is a large software environment with many disparate components. The primary ones are:

- The ICL language (in which the test suite programs are written) and extensions which contain emulation libraries to support optional hardware,
- The test suite, including different versions of architecture dependent application functions,
- The virtual machine emulator which runs the ICL programs and generates traces,
- The trace compiler, which takes virtual machine traces and architectural specifications as input and outputs target machine code and memory reference traces,
- The cache simulator, which takes as input memory reference traces and cache parameters and outputs cache behavior,
- The communication network simulators for circuit switched and packet switched models, and
- The virtual machine code to target machine code dictionary generator which is used to expand virtual machine instructions into target machine code.

We have demonstrated the usefulness of ENPASSANT, first in evaluating the effects of varying individual parameters and features, and second, by showing the usefulness in examining trade-offs in overall MPA system design. Some of the results obtained in evaluating particular components are as follows.

- In the datapath, we showed the effects on fixed and floating point instruction performance of many features and parameters in the PE internals. These included datapath and ALU width, multiply circuits, and floating point support such as barrel shifters.
- In the memory hierarchy we showed the size of register file necessary to hold the working sets for the various application codes. We also showed the cache size required to be effective and the surprising result that the smaller the cache block size, the better the performance.
- In the communication networks, we again showed a surprising result for a class of non-uniform routing patterns with high congestion: adding bi-directional transfer capability across links doubles performance.

Some of the results obtained by examining system issues are as follows.

- In the current CAAPP configuration, increasing the PE ALU and datapath widths alone has relatively small benefit: the speed up is bounded at a factor of about 30%.
- Another level of memory is extremely useful, almost essential, in reducing the proportion of time required for memory access as the virtualization factor increases.

12.2 Contributions

The primary contribution of this work is the building of an environment, ENPASSANT, for the evaluation of MPA architectures with respect to real program executions. The virtues of the system were described in the previous section. Of course building a system, no matter how large, does not by itself indicate that research has been done. In this case, however, a system has been constructed with capabilities that are in many respects considerably beyond what is currently available. The uniqueness of ENPASSANT is that it provides a level of detail comparable to a detailed simulator for a particular target machine, yet at the same time is flexible enough to allow the simulation of a wide variety of such target machines. ENPASSANT is also substantially faster than a detailed simulator allowing much more of the MPA architectural design space to be examined than was previously possible.

Building ENPASSANT has required solving several significant problems; some of these solutions are themselves significant contributions. The most important of these is the MPA virtual machine methodology, which is the basis for the efficiency and flexibility of ENPASSANT. One of the key components, virtual machine emulation, was found to be more than an order of magnitude faster than detailed simulation. Virtual machine emulation also allows the architect to use the same application programs for all target machines without recoding or even recompiling. In fact, the virtual machine emulation and trace generation needs to be done only once per application code for any network/array size combination.

Although virtual machine emulation has been used elsewhere, for example in GT-RAW, the version used by ENPASSANT has a unique set of capabilities. Among the many differences are that GT-RAW is based on execution-driven simulation, while ENPASSANT is basically trace driven; and that GT-RAW supports multiprocessors as well as MPAs, while ENPASSANT supports MPAs at a greater level of detail. Also, ENPASSANT takes advantage of the characteristic mode of control used by SIMD processors to accelerate the virtual machine emulation.

The virtual machine methodology, in turn, required other subproblems to be solved. Several of these solutions are important contributions as well.

- A language needed to be created that could be used to write programs that could be used to simulate all the target machines. This was accomplished by extending ICL and integrating function libraries that emulate optional hardware. We know of no other language or language extension with this capability.

- The issue of fairness was addressed through the inclusion of application function libraries with the test suite. Although function libraries are common, ones with multiple functions depending on the availability of hardware are not. We believe our experience with selective recoding adds a significant data point to the study of task portability among massively parallel processors.

- The trace compiler enables target machine specific behavior to be reconstructed from the virtual machine trace and the architectural specification. The trace compiler generates virtual PE emulation code, assigns registers, and generates target machine code. We believe this idea to be unique.

There are many other contributions that arose from the creation of ENPASSANT. Among them are the assembly of a suite of non-trivial programs having diverse arithmetic, communication, and memory requirements. Together, these codes provide a more complete test bed for MPAs than was previously available.

Perhaps an even more important set of contributions than those just described will be the recommendations that ENPASSANT has already and will continue to help provide for the construction of the next generation MPAs. Although MPAs have been designed since the 1950's and built since the early 1960's, there were clearly many important properties which still needed to be determined at the start of this study. It is likely that there are many more surprising results waiting to be found. Among the most significant determinations made so far are the following.

- PE cache is critical if MPAs are to maintain cost-effective performance in the face of high virtualization factors. A relatively small cache, on the order of a few hundred bytes per PE and only 5 to 10 times larger than the register file, is necessary to prevent memory from becoming the bottleneck. Naturally, these numbers may not hold for database and other memory intensive applications.

- The cache block size should be as small as possible. Also, direct mapped caches need to be more than twice the size of fully associative caches to achieve equivalent performance.

- In k-ary n-cube packet switched networks, higher dimensionality does not seem to provide a significant benefit in the machine vision domain. Also, support for bi-directional transfer should be considered essential if the target MPA will be required to route dense non-uniform reductions.

12.3 Future Work

The first item of future work is to continue to use ENPASSANT to generate results: there are many combinations of design features that still need to be examined.

MPAs are useful in more domains than the spatially mapped applications examined here. An architectural study should be undertaken with respect to, for example, linear algebra applications. This will require extending ICL as described in the next section.

ENPASSANT is still a prototype system. The efficiency of the trace compiler can probably be improved by at least a factor of 2 and perhaps much more than that. Since traces need to be generated relatively rarely, the trace compiler is currently the bottleneck.

We are currently generating results from ENPASSANT on (what is now) antiquated hardware. Porting to a faster system with, especially, more disk space is a priority. This, together with the efficiencies we expect to gain, should enable the examination of target machines that use large virtualization factors.

ENPASSANT does not support all the features we would like. For example, we will be adding support for local address autonomy (see below) and for more complex PEs, including those with (at least) simple pipelines. Also, there is an entire aspect of MPA design that is not covered—the issue of breaking synchrony to enable higher clock rates. This includes more local control decoding to reduce the global issue rate needed.

The empirical results obtained about the packet switched network give clues as to what future network designs should look like: a balance must be achieved between alleviating congestion and following the 'physics' of the routing pattern.

Eventually, we would like to combine ENPASSANT with a system that automatically gauges the cost of various components. Such a system has been developed for micro-processors [14]; developing a similar system for MPAs could be significantly simpler because of the high degree of replication typical in these processors.

Finally, we hope to create a complete MPA design and to (at least) see a significant number of our recommendations be integrated into a production MPA.

12.4 Extensions to ENPASSANT

ENPASSANT is limited by the components that the user is allowed to select in the architectural specification. Although we have included a wide range of features and parameters, they hardly consist of a complete description of the design space. In this section we describe the general issues involved in extending ENPASSANT and the specific methods for three cases.

We partition the possible ENPASSANT extensions into two broad categories: new architectural features and new ways of using existing features. The first category includes adding support for new components such as for a router network that does not fall into the four models currently supported. The second category requires adding support for a desired construct that is not currently available; for example, a parallel data type having more than 2 dimensions.

In general, extensions to ENPASSANT require the following:

- The extension must be supported by ICL. That is, the programmer must be able to write programs that use the new feature appropriately. If ICL as it stands does not have the appropriate semantics, then they must be added. This can involve adding a data type, adding an instruction, or expanding the scope of an operator.

- The new language constructs supporting the new feature will not be supported directly on machines without that feature. For those machines, the new feature must be emulated.

- If the new feature causes different algorithms for tasks in the test suite to be optimal, then those portions of the tasks must be recoded and added to the AFL.

- The new language construct must be executable on the host machine to support virtual machine emulation.

- Additions may need to be made to the evaluator.

We now look at three likely extensions.

ICL currently supports only a 2 dimensional parallel data type, the Plane. Although the Plane is extremely useful in the spatially mapped applications we have examined and maps particularly well onto MPAs, there is no reason that ICL cannot be extended to support, say, the Shape type found in C* [138]. The Shape allows the user to specify parallel data types with up to 31 dimensions, although the sizes of those dimensions must be fixed as with the Plane. The following actions must be taken.

- The language must be extended to include the new class so that programs can be expressed using the new type.
- The virtual machine emulator must be extended. This mostly entails giving routines that currently deal with 2 dimensional arrays the capability of also handling arrays with up to 31 dimensions.
- The trace compiler currently handles virtual PE emulation. These routines must be extended to take care of the mappings of the shapes to various size arrays. Also, shape alignment must be handled there. These problems are not trivial, but they are well understood [91, 85, 86].

ENPASSANT currently does not support local indexing. Local indexing, or local address autonomy is the capability of particular PEs to access data that is offset by an amount in a local register. This gives the programmer's model support for Planes of stacks, queues, and pointer structures. Local indexing is useful, but not as essential in MPAs as in serial processing. MPA parallel variables already contain thousands of elements; structures with thousands of stacks are certainly conceivable, say, in database applications, but hardly ubiquitous. In particular, local indexing was not missed in the creation of the application suite, and variable lists were not found to be a help in routing ([70]). However, it is easy to conceive of applications where it would be essential, and support for local indexing would certainly make programming MPAs more convenient. The following actions must be taken.

- Constructs must be added to ICL. This can either be in the form of general support, as in MPL [104], or more restricted version. In the latter case, simply giving the user the capability of using a Plane as an index for an array of Planes would give the user a FORTRAN-77 type of capability.
- The addition to the virtual machine emulator would be simple: routines that currently operate on arrays would need to be able to operate on arrays of pointers.
- An emulation must be added for the indexed references.
- Routines that are recoded to take advantage of local indexing must have versions provided for which it is not required.

- Modeling the interaction of local address autonomy and PE cache will cause some complications. But this is to be expected since hardware that supports such a combination would necessarily be complex. In particular, if frequent indexed memory references were made and the indexes were unbounded, then separate memory reference traces would need to be captured. In the likelier case where the index is bounded, it should be possible to approximate the performance by saving the range of possible values for each indexed memory access.

Four communication network models are currently supported. Not supported, for example, is any kind of pyramid network. The rationale in that selection is that they are currently not as popular as they were in the mid-1980s. Still, they have been shown to be useful for some applications, such as image compression. The following actions must be taken.

- The particular communication operation inherent in the pyramid—moving up and down levels—is not currently supported by ICL. The appropriate instruction must be added.

- The virtual machine emulator would also need to be augmented to support the new instruction.

- The pyramid movement instruction must be emulated on the other networks.

- Emulations of the communication operations currently available in ICL must be written to take advantage of the pyramid.

- Routines that are recoded to take advantage of pyramid operations must have versions provided for which those operations are not required.

- The evaluator would require little change.

From the three examples of extensions just described, it is apparent that adding significant new capabilities to ENPASSANT is not trivial; it is likely to require weeks of effort. When one considers the alternatives, however, of building prototypes or dedicated simulators, the time is quite reasonable, and further demonstrates the flexibility of the approach employed in ENPASSANT. We thus expect that ENPASSANT will continue to be a useful architectural tool as it evolves in the future.

Bibliography

[1] ACM SIGARCH, and IEEE Computer Society. *Proceedings of the 19th Annual International Symposium on Computer Architecture.* ACM, New York, NY, 1992.

[2] ACM SIGARCH, and IEEE Computer Society. *Proceedings of the 20th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[3] ACM SIGARCH, and IEEE Computer Society. *Proceedings of the 21st Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[4] Active Memory Technology, Inc. *AMT DAP Series: Technical Overview.* Active Memory Technology, Inc., Irvine, CA 92714, 1988.

[5] Active Memory Technology, Inc. *DAP Applications.* Active Memory Technology, Irvine, CA 92714, 1992.

[6] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[7] Almasi, G. S., and Gottlieb, A. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., Reading, MA, 1989.

[8] Alverson, G. A., and Notkin, D. Program restructuring for effective parallel portability. *IEEE Transactions on Parallel and Distributed Systems 4*, 9 (1993), 1041–1059.

[9] Anandan, P. A computation framework and an algorithm for the measurement of visual motion. *International Journal of Computer Vision 2*, 3 (1989), 283–310.

[10] Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), pp. 108–120.

[11] Annaratone, M., and et al. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers C-36*, 12 (1987), 1523–1538.

[12] Axelrod, T., Dubois, P., and Eltgroth, P. A simulator for MIMD performance prediction: Application to the S-1 MkIIa multiprocessor. *Parallel Computing 1* (1984), 237–274.

[13] Bailey, D. H., and et al. The NAS parallel benchmarks. *International Journal of Supercomputing Applications 5*, 3 (1991), 63–73.

[14] Bakoglu, H. B. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley Publishing Company, Reading, MA, 1990.

[15] Ballard, D. H., and Brown, C. M. *Computer Vision.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

[16] Baron, R. J., and Higbie, L. *Computer Architecture Case Studies.* Addison-Wesley Publishing Company, Reading, MA, 1992.

[17] Barrow, H., and Tenenbaum, J. M. Recovering intrinsic scene characteristics from images. In *Computer Vision Systems*, A. Hanson and E. Riseman, Eds. Academic Press, New York, NY, 1978.

[18] Batcher, K. E. Design of the Massively Parallel Processor. *IEEE Transactions on Computers C-29*, 9 (1980), 836–840.

[19] Batcher, K. E. Bit-serial parallel processing systems. *IEEE Transactions on Computers C-31*, 5 (1982), 377–384.

[20] Belady, L. A. A study of replacement algorithms for a virtual storage computer. *IBM System Journal 5*, 2 (1966), 78–101.

[21] Benes, V. E. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, 1965.

[22] Beveridge, J. R., Griffith, J., Kohler, R. R., Hanson, A. R., and Riseman, E. M. Segmenting images using localized histograms and region merging. *International Journal of Computer Vision 2*, 3 (1989), 311–347.

[23] Bhandarkar, D., and Clark, D. W. Performance from architecture: Comparing a RISC and CISC with similar hardware organization. In *Proceedings of the Fourth International Conference on Architectural Support of Programming Languages and Operating Systems* (1991), pp. 310–319.

[24] Bilardi, G., Pracchi, M., and Preparata, F. P. A critique and an appraisal of VLSI models of computation. In *Proceedings of the CMU Conference on VLSI Systems and Computations* (1981).

[25] Blank, T. The MasPar MP-1 architecture. In *Procedings of the 35th IEEE Computer Society International Conference* (1990), pp. 20–24.

[26] Blelloch, G. E. Scans as primitive parallel operations. *IEEE Transactions on Computers C-38*, 11 (1989).

[27] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures* (1991), pp. 3–16.

[28] Blevins, D. W., Davis, E. W., Heaton, R. A., and Reif, J. H. Blitzen: A highly integrated massively parallel machine. *Journal of Parallel and Distributed Computing 8* (1990), 150–160.

[29] Boldt, M., Weiss, R., and Riseman, E. M. Token-based extraction of straight lines. *IEEE Transactions on Systems, Man, and Cybernetics 19*, 6 (1989), 1581–1594.

[30] Bolotski, M., Amirtharajah, R., Chen, W., Kutscha, T., Simon, T., and Knight, Jr., T. F. Abacus: A high-performance architecture for vision. In *Proceedings of the 12th International Conference on Pattern Recognition* (1994).

[31] Boothe, B. Fast accurate simulation of large shared memory multiprocessors. Tech. Rep. UCB/CSD 92/682, Computer Science Division (EECS), University of California, Berkeley, CA 94720, 1992.

[32] Brady, M. The changing shape of computer vision. *Artificial Intelligence 17*, 1-3 (1981), 1–15.

[33] Brady, M. Computational approaches to image understanding. *Computing Surveys 14*, 1 (1982), 3–71.

[34] Brewer, E. A., Dellarocas, C. N., Colbrook, A., and Weihl, W. E. Proteus: A high-performance parallel architecture simulator. Tech. Rep. MIT/LCS/TR-516, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, 1991.

[35] Brewer, J. E., Miller, L. G., Gilbert, I. H., Melia, J. F., and Garde, D. A single-chip digital signal processing subsystem. In *Proceedings of the 1994 International Conference on Wafer Scale Integration* (1994), pp. 1–8.

[36] Brolio, J., Draper, B. A., Beveridge, J. R., and Hanson, A. R. ISR: a database for symbolic processing of computer vision. *IEEE Computer 22*, 12 (1989).

[37] Broomell, G., and Heath, J. R. Classification categories and historical development of circuit switching topologies. *Computing Surveys 15*, 2 (1983), 95–133.

[38] Burns, J. B., Hanson, A. R., and Riseman, E. M. Extracting straight lines. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8*, 4 (1986), 425–455.

[39] Burrill, J. H. *The Class Library for the IUA: Tutorial.* Amerinex Artificial Intelligence, Inc., Amherst, MA 01003, 1992.

[40] Canny, J. F. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8*, 6 (1986), 679–698.

[41] Cavanagh, J. J. F. *Digital Computer Arithmetic: Design and Implementation.* McGraw-Hill Book Company, New York, NY, 1984.

[42] Clark, D. W. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems 1*, 1 (1983), 24–37.

[43] Clos, C. A study of non-blocking switching networks. *The Bell System Technical Journal 32* (1952), 406–424.

[44] Cloud, E. L. The Geometric Arithmetic Parallel Processor. In *Procedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation* (1988), pp. 373–382.

[45] Cmelik, R. F., Kong, S. I., Ditzel, D. R., and Kelly, E. J. An analysis of SPARC and MIPS instruction set utilization on the SPEC benchmarks. In *Proceedings of the Fourth International Conference on Architectural Support of Programming Languages and Operating Systems* (1991), pp. 290–302.

[46] Computer Science and Telecommunications Board. Academic careers for experimental computer scientists and engineers. *Communications of the ACM 37*, 4 (1994), 87–90.

[47] Covington, R. G., Dwarkadas, S., Jump, J. R., Sinclair, J. B., and Mandala, S. Efficient simulation of parallel computer systems. *International Journal of Computers in Simulation 1* (1991), 31–58.

[48] Covington, R. G., Madala, S., Mehta, V., Jump, J. R., and Sinclair, J. B. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1988), pp. 4–11.

[49] Dally, W. J. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers C-39*, 6 (1990), 775–785.

[50] Dally, W. J., and Seitz, C. L. Deadlock free routing in multiprocessor interconnection networks. *IEEE Transactions on Computers C-36*, 5 (1987).

[51] Danielsson, P. E., and Ericsson, T. S. LIPP—proposals for the design of an image processor array. In *Computing Structures for Image Processing*, M. J. B. Duff, Ed. Academic Press, New York, 1983.

[52] Daumueller, K. Extracting lines with a reconfigurable mesh parallel processor. Tech. Rep. CS-TR-94-59, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, 1994.

[53] Davis, H. M. Multiprocessor simulation: Achieving accuracy, efficiency, and flexibility. Tech. Rep. CSL-TR-93-586, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, 1993.

[54] Davis, R. The ILLIAC IV processing element. *IEEE Transactions on Computers C-18*, 9 (1969), 800–816.

[55] Dubois, M., Briggs, F. A., Patil, I., and Balakrishnan, M. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing* (1986), pp. 909–916.

[56] Duff, M. How not to benchmark image processors. In *Evaluation of Multicomputers for Image Processing*, L. Uhr, K. Preston, Jr., S. Levialdi, and M. J. B. Duff, Eds. Academic Press, Boston, MA, 1986.

[57] Dutta, R. Parallel dense depth maps from motion on the image understanding architecture. In *Proceedings of the 1993 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (1993), pp. 154–159.

[58] Falkoff, A. D. Algorithms for parallel search memories. *Journal of the ACM 9*, 4 (1962), 488–511.

[59] Feng, T. A survey of interconnection networks. *IEEE Computer 14*, 12 (1981), 12–27.

[60] Fisher, J. R., and Dorband, J. Applications of the MasPar MP-1 at NASA/Goddard. In *Proceedings of the 1991 IEEE Computer Conference* (1991), pp. 278–282.

[61] Fleming, P. J., and Wallace, J. J. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM 29*, 3 (1986), 218–221.

[62] Foster, C. C. *Content Addressable Parallel Processors.* Van Nostrand Reinhold Co., New York, NY, 1986.

[63] Fountain, T. J. CLIP4: a progress report. In *Languages and Architectures for Image Processing*, S. L. M. J. B. Duff, Ed. Academic Press, Boston, MA, 1981.

[64] Goldschmidt, S. R., and Hennessy, J. L. The accuracy of trace-driven simulations in multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1993), pp. 146–157.

[65] Grondalski, R. A VLSI chip set for a massively parallel architecture. In *Proceedings of the Solid State Circuits Conference* (1987), pp. 1–10.

[66] Hall, C. B., and O'Brien, K. Performance characteristics of architectural features of the IBM RISC System/6000. In *Proceedings of the Fourth International Conference on Architectural Support of Programming Languages and Operating Systems* (1991), pp. 303–309.

[67] Heidelberger, P., and Lavenberg, S. S. Computer performance evaluation methodology. *IEEE Transactions on Computers C-33*, 12 (1984), 1195–1220.

[68] Hennessy, J. L., and Jouppi, N. P. Computer technology and architecture: An evolving interaction. *IEEE Computer 24*, 9 (1991), 18 – 29.

[69] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[70] Herbordt, M. C., Corbett, J. C., Spalding, J., and Weems, C. C. Practical algorithms for online routing on fixed and reconfigurable meshes. *Journal of Parallel and Distributed Computing 20*, 3 (1994), 341–356.

[71] Herbordt, M. C., and Weems, C. C. Efficient algorithms for parallel-prefix and reduction using coterie structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence under review* (1995).

[72] Herbordt, M. C., Weems, C. C., and Scudder, M. J. Non-uniform region processing on SIMD arrays using the coterie network. *Machine Vision and Applications 5*, 2 (1992), 105–125.

[73] Hillis, W. D. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[74] Hillis, W. D., and Steele Jr., G. L. Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1170–1183.

[75] Holland, J. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. In *Proceedings of the Eastern Joint Computer Conference* (1959), pp. 108–113.

[76] Holmer, B. K., and et al. Fast Prolog with an extended general purpose architecture. In *Proceedings of the 17th International Symposium on Computer Architecture* (1990), pp. 282–291.

[77] Hsu, J.-M., and Banerjee, P. Performance measurement and trace driven simulation of parallel CAD and numeric applications on a hypercube multicomputer. In *Proceedings of the 17th International Symposium on Computer Architecture* (1990), pp. 260–269.

[78] Huang, T. S. Motion analysis. In *The Encyclopedia of Artificial Intelligence*, S. C. Shapiro, Ed. John Wiley and Sons, New York, NY, 1987.

[79] Hunt, D. J. The ICL DAP and its application to image processing. In *Languages and Architectures for Image Processing*, S. L. M. J. B. Duff, Ed. Academic Press, London, 1981.

[80] Irwin, M. J., and Owens, R. M. A two-dimensional, distributed logic architecture. *IEEE Transactions on Computers C-40*, 10 (1991), 1094–1101.

[81] Jackson, J. The Data Transport Computer: a 3-dimensional massively parallel SIMD computer. In *Proceedings of the 35th IEEE Computer Society Conference* (1991), pp. 264–269.

[82] Jonker, P. P. *Morphological Image Processing: Architecture and VLSI Design*. Kluwer, The Netherlands, 1992.

[83] Kahn, P., Kitchen, L., and Riseman, E. M. A fast line finder for vision-guided robot navigation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 3 (1990), 1098–1102.

[84] Kermani, P., and Kleinrock, L. Virtual cut-through: A new computer communication switching technique. *Computer Networks 3* (1979), 267–286.

[85] Knobe, K., Lucas, J. D., and Steele Jr., G. L. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing 8* (1990), 102–118.

[86] Knobe, K., and Natarajan, V. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation* (1990), pp. 416–423.

[87] Kochevar, P. A simple light simulation algorithm for massively parallel machines. *Journal of Parallel and Distributed Computing 13* (1991), 193–201.

[88] Komen, E. R. *Low-level Image Processing Architectures: Compared For Some Non-linear Recursive Neighbourhood Operations*. PhD thesis, Technical University Delft, Delft, The Netherlands, 1990.

[89] Krikelis, A. Computer vision applications with the Associative String Processor. *Journal of Parallel and Distributed Computing 13* (1991), 170–184.

[90] Lang, D. E., Agerwala, T. K., and Chandy, K. M. A modeling approach and design tool for pipelined central processors. In *Proceedings of the International Symposium on Computer Architecture* (1979), pp. 122–129.

[91] Lawrie, D. H. Access and alignment of data in an array processor. *IEEE Transactions on Computers C-24*, 12 (1975), 1145–1155.

[92] Leighton, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman Publishers, San Mateo, CA, 1992.

[93] Li, H., and Maresca, M. The Polymorphic-Torus Architecture for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11*, 3 (1989), 233–243.

[94] Li, H., and Maresca, M. Polymorphic Torus Network. *IEEE Transactions on Computers C-38*, 9 (1989), 1345–1351.

[95] Ligon III, W. B., and Ramachandran, U. An empirical methodology for exploring reconfigurable architectures. *Journal of Parallel and Distributed Computing 19* (1993), 323–337.

[96] Ligon III, W. B., and Ramachandran, U. Simulating interconnection networks in RAW. In *Proceedings of the 7th International Parallel Processing Symposium* (1993), pp. 268–275.

[97] Lilja, D. J. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *Computing Surveys 25*, 3 (1993), 303–338.

[98] Little, J. J., Blelloch, G. E., and Cass, T. A. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11*, 3 (1989), 244–257.

[99] Marek, T. C. *Comparative Evaluation of Processor Architectures for Massively Parallel Systems.* PhD thesis, North Carolina State University, 1992.

[100] Marek, T. C., and Davis, E. W. Quantitative studies of processing element granularity. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation* (1992), pp. 551–552.

[101] Marr, D. *Vision.* W.H. Freeman, San Francisco, CA, 1982.

[102] Marr, D., and Hildreth, E. Theory of edge detection. *Proceedings of the Royal Society of London B 207* (1980), 187–217.

[103] MasPar Computer Corporation. *MasPar MP-1 Principles of Operation.* Sunnyvale, CA, 1990.

[104] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL).* Sunnyvale, CA, 1990.

[105] MasPar Computer Corporation. *The Solutions Leader in Massively Parallel Computing.* Sunnyvale, CA, 1992.

[106] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[107] McCormick, B. T. The Illinois Pattern Recognition Computer – ILLIAC III. *IEEE Transactions on Electronic Computers C-12*, 12 (1963), 791–813.

[108] Nickolls, J. R. The design of the MasPar MP-1: a cost effective massively parallel computer. In *Procedings of the 35th IEEE Computer Society International Conference* (1990), pp. 25–28.

[109] Nutt, G. J. A case study of simulation as a computer system design tool. *IEEE Computer 11*, 10 (1978), 31–36.

[110] Ohlander, R., Price, K., and Reddy, D. R. Picture segmentation using a recursive region splitting method. *Computer Graphics and Image Processing 8* (1978), 313–333.

[111] Overton, K. J., and Weymouth, T. E. A noise reducing preprocessing algorithm. In *Proceedings of the Conference on Pattern Recognition and Image Processing* (1979), pp. 498–507.

[112] Owens, R. M., Irwin, M. J., Nagendra, C., and Bajwa, R. S. Computer vision on the MGAP. In *Proceedings of the Workshop on Computer Architectures for Machine Perception* (1993), pp. 337–341.

[113] Parkinson, D., and Jesshope, C. R. The AMT DAP 500. In *Proceedings of the 33rd IEEE Computer Society Conference* (1988).

[114] Pass, S. The GRID parallel computer system. In *Image Processing System Architectures*, M. J. B. D. J. Kittler, Ed. John Wiley & Sons, New York, NY, 1985.

[115] Patt, Y. Experimental research in computer architecture. *IEEE Computer 24*, 1 (1991), 14–16.

[116] Pease, D., and et al. PAWS: a performance evaluation tool for parallel computing systems. *IEEE Computer 24*, 1 (1991), 18–29.

[117] Potter, J. L., Ed. *The Massively Parallel Processor.* The MIT Press, Cambridge, MA, 1985.

[118] Prasanna Kumar, V. K., and Reisis, D. Image computations on meshes with multiple broadcast. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-11*, 11 (1989), 1194–1202.

[119] Prechelt, L. Measurements of MasPar MP-1216A communication operations. Tech. Rep. Technical Report 01/93, Universitaet Karlsruhe, Karlsruhe, Germany, 1993.

[120] Preston, K. The Abington Cross benchmark survey. *IEEE Computer 22*, 7 (1989), 9–18.

[121] Reeves, A. P. A systematically designed binary array processor. *IEEE Transactions on Computers C-29*, 4 (1980), 278–287.

[122] Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. The Wisconsin Wind Tunnel: virtual protoyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1993), pp. 48–60.

[123] Rosenfeld, A. Image analysis: Problems, progress and prospects. *Pattern Recognition 17*, 1 (1984), 3–12.

[124] Rosenfeld, A. A report on the DARPA Image Understanding Architectures Workshop. In *Proceedings: Image Understanding Workshop* (1987), pp. 298–301.

[125] Schmitt, L. A., and Wilson, S. S. The AIS-5000 parallel processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-10*, 3 (1988), 320–330.

[126] SGS-Thompson Microelectronics Group. *The T9000 Transputer Products Overview Manual*, 1991.

[127] Siegel, H. A model of SIMD machines and a comparison of various interconnection networks. *IEEE Transactions on Computers C-28*, 12 (1979), 907–917.

[128] Smith, A. J. Cache memories. *Computing Surveys 14*, 3 (1982), 473–530.

[129] Smith, J. E. Characterizing computer performance with a single number. *Communications of the ACM 31*, 10 (1988), 1202–1206.

[130] Snyder, L. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science 1* (1986), 289–317.

[131] Stone, H. S. *High-Performance Computer Architecture.* Addison-Wesley, Reading, MA, 1987.

[132] Strong, J. Computations on the Massively Parallel Processor at the Goddard Space Flight Center. *Proceedings of the IEEE 79*, 4 (1991), 548–558.

[133] Stunkel, C. B., and Fuchs, W. K. TRAPEDS: producing traces for multicomputers via exectution driven simulation. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1989), pp. 70–78.

[134] Sunwoo, M. H., and Aggarwal, J. K. A sliding memory plane array processor. *IEEE Transactions on Parallel and Distributed Systems 4*, 6 (1993), 601–613.

[135] Systems Performance Evaluation Cooperative. *SPEC Newsletter: Benchmark Results.* Waterside Associates, Freemont, CA, 1990.

[136] Tanimoto, S. Memory systems for highly parallel computers. *Proceedings of the IEEE 79*, 4 (1991), 403–415.

[137] Thinking Machines Corporation. Connection Machine model: CM-2 technical summary. Tech. Rep. T.R. HA87-4, Thinking Machines Corporation, Cambridge, MA, 1987.

[138] Thinking Machines Corporation. *C* Programming Guide.* Cambridge, MA, 1990.

[139] Tsotsos, J. K. A 'complexity level' analysis of immediate vision. *International Journal of Computer Vision 1*, 4 (1988), 303–320.

[140] Tucker, L. W. Architecture and applications of the Connection Machine. *IEEE Computer 21*, 8 (1988), 26–38.

[141] Tucker, L. W., Feynman, C. R., and Fritzsche, D. M. Object recognition using the Connection Machine. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (1988), pp. 871–878.

[142] Unger, S. H. A computer oriented toward spatial problems. *Proceedings of the IRE 47* (1958), 1744–1750.

[143] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[144] Weems, C. C. *Image Processing on a Content Addressable Array Parallel Processor.* PhD thesis, University of Massachusetts, Department of Computer and Informaction Science, University of Massachusetts, Amherst, MA 01003, 1984.

[145] Weems, C. C. Architectural requirements of image understanding with respect to parallel processing. *Proceedings of the IEEE 79*, 4 (1991), 537–547.

[146] Weems, C. C., and Burrill, J. R. The Image Understanding Architecture and its programming environment. In *Parallel Architectures and Algorithms for Image Understanding*, V. Prasanna Kumar, Ed. Academic Press, Orlando, FL, 1991.

[147] Weems, C. C., Levitan, S. P., Hanson, A. R., Riseman, E. M., Nash, J. G., and Shu, D. B. The Image Understanding Architecture. *International Journal of Computer Vision 2*, 3 (1989), 251–282.

[148] Weems, C. C., Riseman, E. M., Hanson, A. R., and Rosenfeld, A. The DARPA image understanding benchmark for parallel computers. *Journal of Parallel and Distributed Computing 11* (1991), 1–24.

[149] Weiker, R. An overview of common benchmarks. *IEEE Computer 23*, 12 (1990), 65–75.

[150] Wilding, N. B., Trew, A. S., Hawick, K. A., and Pawley, G. S. Scientific modeling with massively parallel SIMD computers. *Proceedings of the IEEE 79*, 4 (1991), 574–585.

[151] Williams, E., and Bobrowicz, F. Speedup predictions for large scientific parallel programs on Cray X-MP-like architectures. In *Proceedings of the 1985 International Conference on Parallel Processing* (1985), pp. 541–543.

[152] Zucker, S. Early vision. In *Encyclopedia of Artificial Intelligence*, S. C. Shapiro, Ed. John Wiley and Sons, New York, NY, 1987.