

ENPASSANT: AN ENVIRONMENT FOR EVALUATING MASSIVELY PARALLEL ARRAY ARCHITECTURES FOR SPATIALLY MAPPED APPLICATIONS*†

MARTIN C. HERBORDT‡

*Department of Electrical and Computer Engineering
University of Houston, Houston, TX 77204, USA*

CHARLES C. WEEMS

*Department of Computer Science
University of Massachusetts, Amherst, MA 01003, USA*

Although massively parallel arrays for spatially mapped applications have been proposed since the 1950s⁴² and built since the 1960s,¹² there have been very few systematic empirical studies that cover more than a small fraction of the design space. The problems have included the lack of a test suite of non-trivial application codes; inadequate language support; the difficulties of balancing evaluation performance with flexibility; and balancing test suite portability with accuracy of evaluation. We describe an environment that addresses these problems. A realistic workload including a series of applications currently being used as building blocks in vision research has been constructed. Both flexibility in architectural parameter selection and simulation efficiency are maintained with a novel new technique that combines virtual machine emulation with trace-driven simulation. The trade-off between fairness to diverse target architectures and programmability of the test suite is addressed through the use of operator and application libraries for a small set of critical functions. We also present examples of the type of results we are obtaining, including the effects of changing ALU designs and datapath widths, finding critical points in register set and cache sizes, the benefits of various types of router networks, and the performance cost of processor virtualization.

Keywords: Architectural evaluation, SIMD arrays, massively parallel processing, low-level vision, intermediate-level vision, architectures for vision, trace-driven simulation.

1. INTRODUCTION

Computer vision is among the most computationally intensive tasks: Estimates have been made that a rate of execution several orders of magnitude higher than that currently available will be needed to perform real-time image understanding.⁴⁶ The only way such computation rates are physically possible is by using massively parallel processors. Other applications use a similar style of computation and also have very high performance requirements. These include some VLSI layout algorithms

*This work was supported in part by the Defense Advanced Research Projects Agency under contract DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Lab.; under contract DAAL02-91-K-0047, monitored by the U.S. Army Harry Diamond Lab.; and by a CII grant from the National Science Foundation (CDA-8922572).

†A preliminary version of this work was presented at Computer Architectures for Machine Perception, CAMP'93.

‡M. C. Herbordt was supported in part by an IBM Fellowship.

and relaxation-based partial differential equation (PDE) solvers. The question this research addresses is how to make architectural decisions for these types of machines on spatially mapped computations. As there has been some confusion as to what is low-, intermediate- and high-level vision, we instead refer to as *spatially mapped* those tasks that use pixel-PE mappings during a significant part of the computation. In particular, we describe a methodology and software system architecture that analyzes real program executions to evaluate massively parallel array architectures.

Empirical computer architecture research requires the collection of performance data of a workload on sample architectures, i.e. the evaluation of points in the architectural design space. In general there is a choice between obtaining detailed information about a small number of architectures (usually one or two), or doing high-level studies on a class of architectures. Literature on architecture is filled with examples of the former, the latter approach was the basis for the current work on the class of architectures that became known as RISC machines; this approach is described by Hennessy and Patterson in Ref. 16.

The analysis of massively parallel array (MPA) architectures for spatially mapped applications 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 analyzing particular classes of algorithms (e.g. Refs. 11, 20, 21), requirements analysis (e.g. Refs. 41, 46), or feedback from benchmarks (e.g. Ref. 44). 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.

Two systems that have previously employed empirical techniques for MPA architectural analysis are the SIMD Simulator Workbench^{26,27} and GT-RAW.^{23,34} The first of these was developed to examine MPA PE design trade-offs. It is based on a detailed simulator of the BLITZEN⁵ which has been parameterized to enable the examination of multiple granularities (the trade-off between PE ALU width and number of PEs), support for multiplication and floating point instructions, and support for local indexing. The second system, GT-RAW, was developed to analyze reconfigurable architectures, so multiprocessor simulation is emphasized. The overall system design is an execution-driven simulator where the host is a virtual machine. As the programs are executed, the instructions in the different streams are synchronized, communication simulated, and the timing evaluated.

Both systems represent significant advances 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 the following problems.

1. **Computational Intractability.** The simulation of massively parallel arrays requires orders of magnitude more processing than the simulation of a serial processor. The classic choice in design studies of a given level of accuracy are between prototyping (efficient, but inflexible), and simulation (flexible, but very slow). Previous benchmark studies have been of the former kind and thus have neither explored a large part of the architectural design space, nor investigated the effects of varying multiple parameters simultaneously. Simulation studies have also only studied a small part of the design space and generally at a coarse level of detail.

2. **Programmability.** Porting code among massively parallel architectures (or potential designs) sometimes requires that functions be recoded to use different algorithms. Otherwise the appropriate features will not be properly used and the results will be skewed.^a 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 have leaned towards being task oriented^{34,36,47} 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.

3. **Appropriate Workload.** 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. Ref. 47).

We address these issues as follows. Flexibility, while maintaining enough efficiency to explore a significant part of the design space, is achieved by combining virtual machine emulation — that is, minimal behavioral simulation that generates traces of virtual machine code — with trace-driven simulation. The emulation is orders of magnitude faster than detailed simulation, while the trace-driven simulation enables flexible analysis. 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. We address the problem of proper workload selection by including applications in our test suite that are in continual use in a machine vision research environment, as well as sample codes from VLSI layout and a PDE solver. These programs have the size and complexity to appropriately exercise sample designs.

^aThe porting 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.

The primary result that we present in this paper is an evaluation environment that is accurate, flexible, efficient, fair, and programmable. We call it ENPASSANT (ENvironment for PARallel System Simulation ANALysis Tools). We demonstrate this with numerous examples, including a case study evaluating register/cache tradeoffs with respect to different processor virtualization factors (due to varying the size ratio of the data array to the processor array).

The rest of this paper is organized as follows. The next section presents the application and architecture domains. There follows an overview of the simulator architecture. The next two sections describe some details of the simulator components. We close with our case study and conclusion.

2. THE ARCHITECTURE AND APPLICATION DOMAINS

2.1. Project Goals

The overall goal of any simulation study is to provide data that will be the basis for design decisions. Particular questions we wish to address include:

1. Is there a system level bottleneck? What is the relative importance of improving performance on current designs of inter-PE communication, the PE memory hierarchy, and the PE internals?
2. Given a balanced design, what are the minimum attributes for an ‘effective’ design, i.e. one that is competitive with candidates from other architectural classes?
3. What are the relative benefits of increasing the processor array size (number of PEs) versus increasing PE complexity?
4. What are the effects of particular design choices? For example, how does changing the machine from a one operand to a two or a three operand machine affect the datapath/ALU performance? Is it more worthwhile to increase the datapath widths? What is more important, adding registers, or adding another level to the memory hierarchy? What is more important for a packet switched router network, increasing its dimensions (decreasing its diameter) or increasing the bandwidth between pairs of nodes?
5. What is the limit of ‘obvious’ improvements, e.g. increasing the datapath width? What functions are best hardwired? What hardware is best parallelized?

2.2. The Test Suite

The primary purpose of any test suite is to reflect the workload likely to be encountered in the domain being studied. We have two criteria in selecting the tasks.

The first is to make sure that the tasks, as a suite, contain a representative sample of most of the *types* of computations found in the domain of spatially mapped computations (span the computation space). This has been done to ensure that all critical components of the designs are exercised fully: even if the weight of a particular type of computation does not precisely match that in a true workload, no significant architectural deficiency should go undetected.

The second criterion is to include applications currently in use. There are at least three reasons for this: (1) the programs must be 'big' enough to realistically exercise the memory hierarchy; (2) small changes in the proportion of expensive instructions (e.g. reduction) can cause a large change in apparent performance; and (3) the 'messy' connecting code, e.g. moving and aligning data, that is a significant part of most computations must be accurately represented. See Tables 1 and 2 for the vision-based test suite elements and the computations they represent. Other test suite programs are a VLSI path routing algorithm, and a relaxation-based PDE solver.

Table 1. Description of vision-based test suite programs.

Application	Description and comments
DARPA IU Benchmark II	Synthetic recognition task developed to evaluate complete image understanding systems. ⁴⁷ We use low-level bottom-up processing and intermediate-level processing components.
Weymouth-Overton preprocessor	Information preserving image filter. Uses edge-model curve fitting. ³¹
Fast line finder	Based on Burns's algorithm ⁷ : segments image by gradient orientation and fits line to resulting regions.
Depth from motion	Computation dominated by correspondence-based matching. ¹⁵
Boldt's line finder	Perceptual organization based. Iterative grouping algorithm. ⁶
Region segmentation	Based on Nagin-Kohler system. ³ Combines histogram-based image splitting and region merging techniques.

Table 2. Broad classes of array computations found in spatially mapped applications, the particular application tasks in which they are used, and the test suite program where they are represented.

Type of computation	Applications where used	Test suite program where used
Pixel and integer array operations	all applications	all test suite programs
Floating point array operations	image preprocessing motion	Weymouth-Overton preprocessor DARPA IU Benchmark II
Window-based communication: small windows	edge detection image filtering	all test suite programs
Window-based communication: large windows	correspondence problem	Depth from motion
Non-uniform communication	grouping segmentation	Boldt's line finder Fast line finder Region segmentation
Non-uniform reduction	segmentation	Region segmentation
Use of non-trivial data structures	grouping segmentation	Boldt's line finder
Internal data movement and alignment	all applications	DARPA IU Benchmark II

2.3. The Architectural Design Space

The design space is the class of architectures known as massively parallel arrays (MPAs), many of which were designed specifically for spatially mapped computations. Examples include the Goodyear MPP,² the CLIP-4,¹⁴ the Cambridge Parallel Processors DAP,³² the Thinking Machines CM-1, CM-2, and CM-200,⁴⁰ the UMass/Hughes CAAPP,⁴⁴ the BLITZEN,⁵ and the MasPar MP-1 and MP-2.⁴

The architectural design space has two sets of components: the first is the set of features common to the entire class of architectures, the second consists of those components that will be varied and evaluated.

We describe the common part first. The architectures all have a number of processing elements on the order of the sizes of the input images. With current technology, this necessitates SIMD control. PEs are all assumed to have a simple ALU, some registers, and some memory. The processor arrays have inter-PE communication networks at least as powerful as a nearest-neighbor mesh. Feedback from array to controller is provided by a global-OR circuit.

We partition the optional part of the architectural design space into features and parameters. These can generally be thought of as switches and dials on an instrument panel, respectively. The distinction was inspired by Snyder's work³⁸ and is discussed in Ref. 18. Roughly speaking, a *parameter* is a component for which a reasonable compiler could be expected to make optimization decisions without user input; the opposite is true of *features*. In general, most components of serial processors are parameters. These include the number of registers, the width of the datapath, the size of the cache, etc. These components are parameters in MPAs as well; another MPA parameter is the number of PEs in the array.

In MPAs, to a much greater extent than in serial processors, there are also components whose presence or absence can cause different algorithms to be optimal. For example, we refer to the change of the inter-PE routing network from a broadcast mesh to a packet switched hypercube as a change in architectural feature. This is because, for several tasks (e.g. connected components, convex hull), these networks have different optimal algorithms. See Fig. 1 for an example of how various popular MPAs can be viewed as collections of features.

3. SIMULATOR FLEXIBILITY AND EFFICIENCY: THE VIRTUAL MACHINE METHODOLOGY

The central issue in machine evaluation is the tradeoff between flexibility, i.e. the ability to view as much of the design space as possible, and efficiency, i.e. the ability to generate quickly significant results for each point that is examined.

At one extreme is the software simulation of complete machines (see e.g. Ref. 1). A simulator can be designed (at least in theory) with enough generality to simulate an arbitrary set of features and with enough detail to be able to simulate any level of granularity. In practice, however, there are two limitations. The first is the complexity of the simulator; simply coding all the options and verifying their correctness is an extremely labor intensive task. The second problem is more

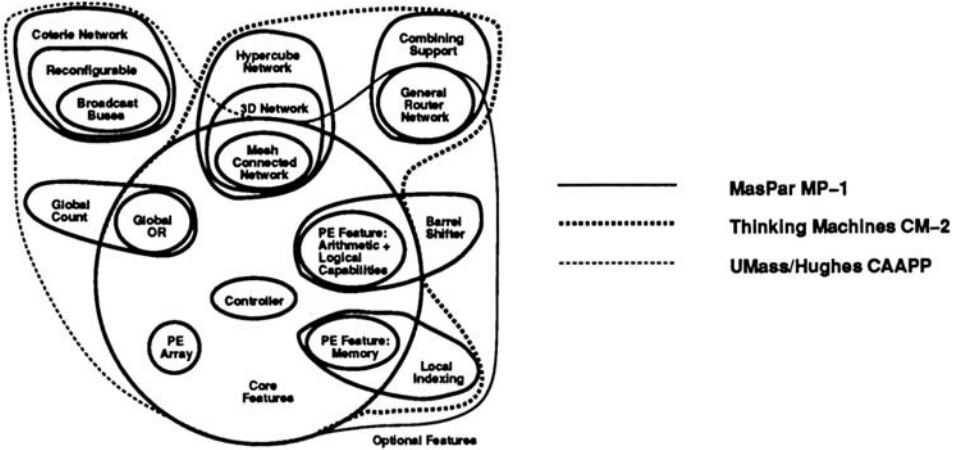


Fig. 1. Representation of three massively parallel arrays (MPAs) as collections of core and optional architectural features.

critical: simulating beyond the coarsest level of detail is extremely time consuming. Complete simulation is generally used either for high-level design (where detail is sacrificed for reduced execution time), as a final check before building a prototype (where every detail is thoroughly checked, but with a minimum amount of test code), or as a software development tool while waiting for hardware to become available.

At the other extreme we have direct machine evaluation (see e.g. Ref. 9). The advantages of this approach are the detail and accuracy of the measurements: there are many problems that never become apparent unless they are observed in a working system. The disadvantages are that one needs a working system, instrumentation can be complex if not designed into the machine, and it is difficult to change parameters. Direct evaluation is often used to measure and tune the performance of existing machines.

Trace-driven simulation has been used for examining the performance of prospective memory hierarchy and pipeline designs (see e.g. Ref. 37 and 22 respectively). Either a memory reference or an instruction trace is generated, usually from an existing system whose device characteristics (other than those being tested) are similar to those of the target machine. In the case of memory evaluation, the memory reference trace can be processed efficiently for a number of parameters simultaneously by using techniques developed in Ref. 28. A disadvantage of trace-driven simulation is that different traces must be generated if processor parameters, such as the size of the register file are varied.

In execution-driven simulation, the test program is executed directly on the host machine, or on one processor of a multiprocessor (see Refs. 10, 35). Only interprocessor communication and/or distributed memory access are simulated. Execution-driven simulation is most useful for evaluating multiprocessor network designs.

None of these methods, however, are appropriate for our use. We do not have an existing machine, nor one to which we would like to restrict our designs, eliminating direct machine evaluation and execution-driven simulation. The design space we are interested in searching is too large for complete machine simulation. And since we are interested in evaluating more than just the memory hierarchy simple trace-driven simulation is also inadequate.

Our method addresses the flexibility/efficiency issue by using a multi-stage process. The basic idea is to run the test programs on a virtual machine emulator, generate a trace, transform the trace with respect to a given architecture model, and derive performance results. Where our technique differs from common trace-driven simulation is that the emulator does not need to resemble the target architecture in any respect: our virtual machine emulator will run efficiently on any machine that runs our virtual machine language (ICL), which is simply C++ augmented with a parallel class. Naturally, the more the host machine resembles the virtual machine programming model (a generic massively parallel array), the better the performance of the virtual machine emulator.

The test suite programs are all written in ICL and do not need to be rewritten to run efficiently for any MPA target architecture. ICL has similar semantics to other data parallel programming languages.⁸ It is like C*,⁴¹ but with the parallel data type restricted to two dimensions and called a Plane rather than a Shape. ICL contains the standard C types for scalars and Planes and the standard C operations for scalar-scalar, scalar-Plane, and Plane-Plane combinations. ICL also contains operations to handle Plane characterization (reductions), interplane data movement (permutes and scans), and support reconfigurable mesh operations (region formation and PE broadcast).

By excluding target machine implementation details from the virtual machine model, we can separate the program execution (and trace generation) from most of the architectural analysis. This has two benefits.

- The test suite code is run with similar efficiency as code written especially for a serial processor. While this still yields the inherent slowdown in the trace capture/compression process (about a factor of 100), it is approximately 100 times faster than running the ICL code on a detailed MPA simulator. Also, only a fraction of the storage is required.
- Since the program traces are evaluated off-line, they need to be generated much less frequently, i.e. only once for any combination of array size, network, and application code.

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 Fig. 2). At a slightly lower level, ENPASSANT contains four major components: the input

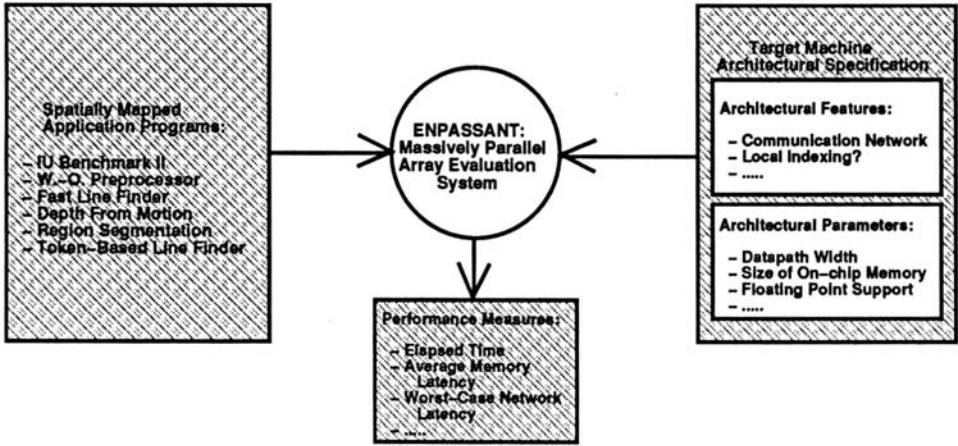


Fig. 2. ENPASSANT system architecture: highest level view.

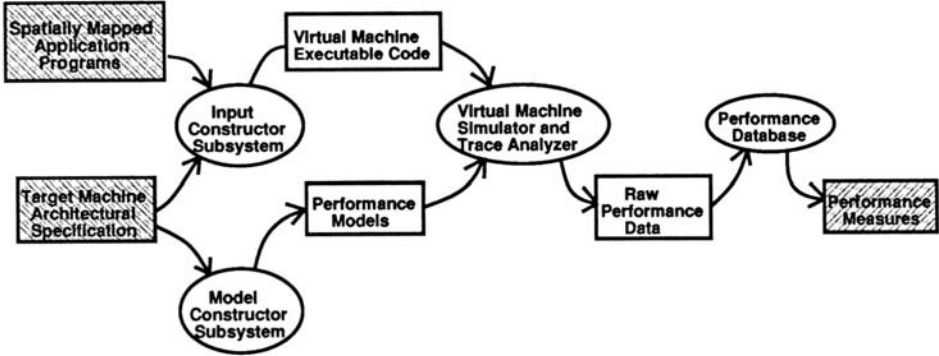


Fig. 3. ENPASSANT system architecture: block diagram.

constructor, the performance model constructor, the virtual machine simulator and trace generator, and the trace analyzer (see Fig. 3).

- The **input constructor** takes as input application programs written in 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 simulator** 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.

4. THE INPUT CONSTRUCTOR SUBSYSTEM

The task of the input constructor is to create efficient code for each possible machine model. See Fig. 4 for its components and flow. Besides the obvious steps of compilation and linking, two issues must be addressed. The first is hardware

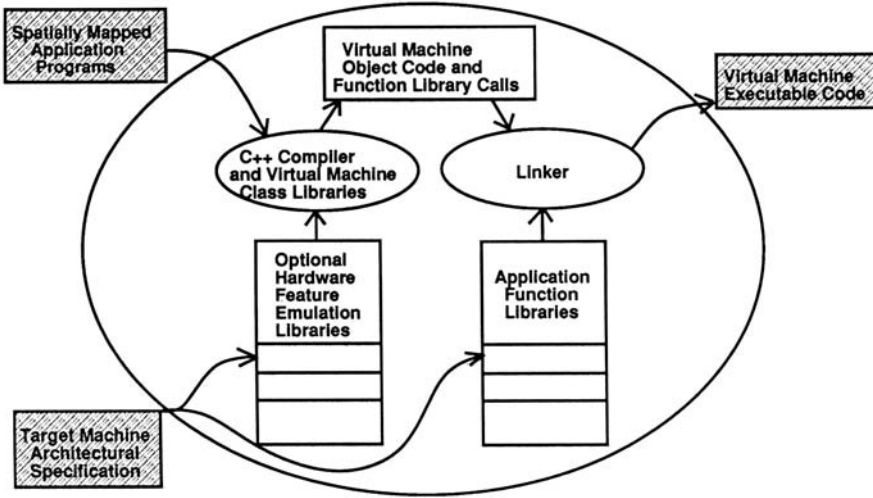


Fig. 4. Input constructor subsystem.

emulation, the second is balancing fairness in evaluation with programmability of the test suite.

4.1. Hardware Emulation

The virtual machine model was designed to present a programmer's model that contains the *union* of the set of hardware features available on *any particular* machine within the domain of massively parallel arrays. Therefore ICL constructs are available for routing, scanning, reduction, global count, and other operations for which hardware support exists on some, but not all machines. Since these virtual machine constructs are not implemented directly on all target architectures, they must be emulated with operations that *are* available.

To do this we have created the Operator Emulation Library (OEL).

For example, the MAX-Scan operation is not supported directly on machines that do not have combining router networks. In order for codes containing those operations to be evaluated with respect to machines not having that particular feature, that operation is replaced with a library call to an emulation function. The set of functions that emulate communication operations is described in Ref. 17.

4.2. Maintaining Fairness and Programmability

Architects of general purpose processors have established the benefits of using trace-driven simulation for architectural evaluation.¹⁶ They have also determined that the criterion for the effective use of this approach is the existence of a portable high-level language for each of the designs being evaluated. Thus generic code for various benchmark suites such as SPEC³⁹ and LINPACK¹³ can run with comparable efficiency on all target platforms for which a reasonable quality compiler exists.

The major difficulty that arises when transferring this method to the domain of massively parallel processors is precisely the difficulty in retaining efficiency while porting code from one processor to another. The reason is not just a question of the existence of a suitable language: it is that algorithms optimal for one parallel architecture are likely to be sub-optimal for other parallel architectures, even within a class of processors such as massively parallel arrays. 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 recognize that this is a critical problem (and the reason for the task oriented nature of most parallel vision benchmarks^{34,36,47}) and address it as follows: sections of the test programs that require different algorithms for efficiency reasons will have them provided.

An example of such a task is labeling connected components. The code used on the CM-2 uses either pointer jumping or segmented-grid-scan based algorithms,²⁵ while that on the CAAPP uses multi-associative leader election via region broadcast.¹⁹ Exchanging or otherwise choosing the inappropriate code would result in the architecture having far worse performance than it is 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 the virtual machine simulator. 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 slow down resulting from using the incorrect algorithm on the CM2-like model was a factor of 13.5. The slow down 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.

5. MODEL GENERATION AND TRACE ANALYSIS

Once the input has been created, it is run on a virtual machine emulator and traces are generated. The traces are then evaluated with respect to three separate components, the datapath and ALU, the memory hierarchy, and the communication network. See Figs. 5 and 6 for the components and data flow in these parts of ENPASSANT.

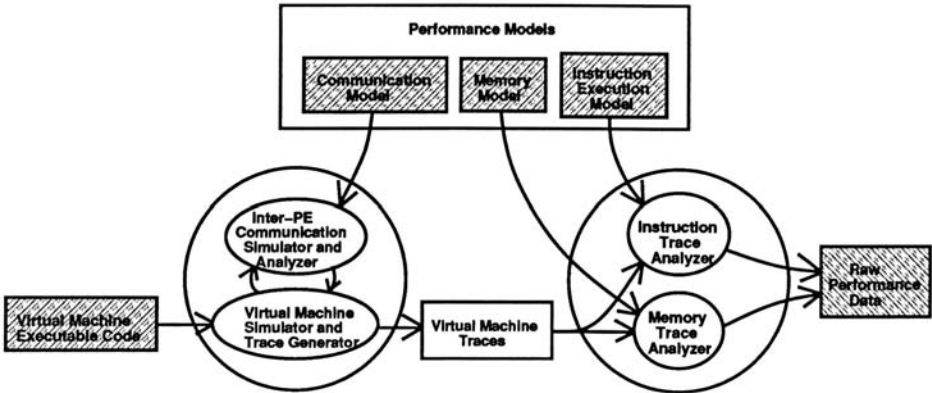


Fig. 5. Virtual machine simulator and trace analyzer.

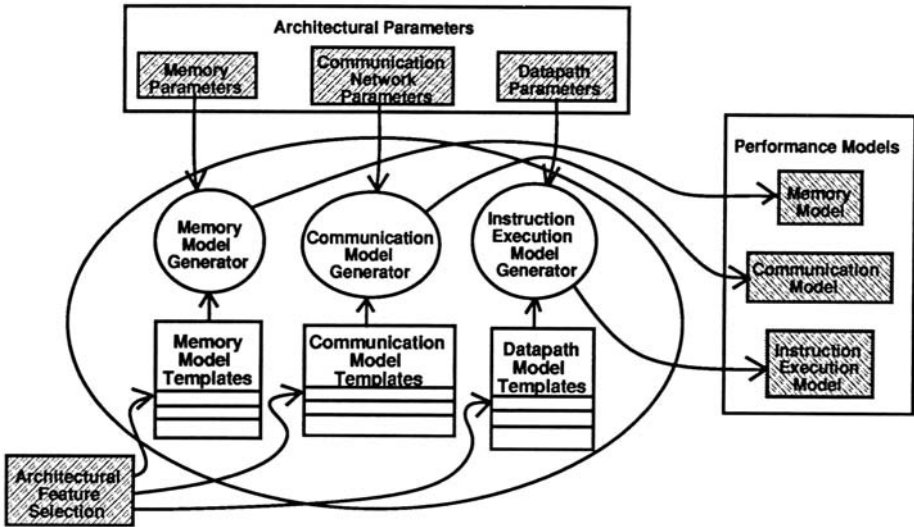


Fig. 6. Model constructor subsystem.

5.1. Evaluating the Datapath

The purpose of the datapath (i.e. the PE ALU and other internals) model generator is to compute timings of the virtual machine instructions executed on the virtual machine emulator. For example, consider a virtual machine instruction that directs the array to add two parallel 32 bit integers and leave the result in the first operand. On the CAAPP, this instruction takes 65 cycles: 1 to clear the carry bit and, being a single operand machine, 2 cycles per bit for the rest of the add. On the MasPar MP-1, this operation takes 8 cycles. The ALU is 4 bits wide, both operands are loaded simultaneously, and the condition codes are handled in parallel.³⁰

The goals of our implementation are (1) to be able to model precisely as many existing datapaths as possible (2) to allow for most conceivable extensions, and (3) to facilitate the combining of different sets of features. Our implementation strategy is to create a *generic* MPA datapath and an accompanying microcode generator for the virtual machine MPA instruction set. The user creates a *particular* datapath model by invoking the model generator, choosing a datapath template from a menu of features, and specifying a set of parameters for those features (see Table 3).

Table 3. Options and their parameters in the datapath microcode generator.

Option	Parameters
Number of register operands per cycle	1,2,3
ALU parameters	Latency, width (1,2,4,8,16,32,64)
Datapath parameters	Latency, width (1,2,4,8,16,32,64, and wider than ALU)
ALU supports bit types	T/F
Both ADD and ADDC instructions	T/F
Shift register	T/F, latency, width (16,32,64,128)
Multiplier	T/F, latency, width (4,8,16,32,64)
Divider	T/F, latency, width (16,32,64)
Barrel shifter	T/F, latency, width (32,64)
Leading one detector	T/F, latency, width (32, 64)
Floating point registers	T/F, width (32,64)
Floating point co-processors	T/F, width (32,64), # of PEs sharing FP unit, load/execution latencies

Table 4. Tasks of the DARPA IU Benchmark II that are used in the test suite program.

Low-level processing
Label connected components
Compute the K -curvature
Extract the corners
Intermediate-level processing
Select components with three or more corners
Find the convex hull of corners of each component
Compute angles between successive corners on hulls
Select corners with K -curvature and computed angles indicating right angle
Label components with three contiguous right angles as candidate rectangles
Compute size, orientation, position, and intensity of candidate rectangles

The model generator uses this information to compute the timings for the virtual machine instruction set. For example, to obtain the behavior of the CAAPP datapath as shown above, the user specifies: 1 register operand per cycle, ALU latency of 1 (one cycle per ALU micro-instruction), ALU width of 1, and that the CAAPP does not support the ADD instruction (i.e. the carry bit must be cleared explicitly).

We now demonstrate datapath design evaluation by presenting simulation results from the bottom-up and intermediate portions of the DARPA IU Benchmark. The constituent tasks can be found in Table 4, see Ref. 47 for details. We use the CAAPP model as a baseline. Besides the characteristics mentioned above, it has an 8-bit wide datapath, supports bit addressing, has 40 bytes of register file, load and store latency of 5 cycles per bit, a 1-bit nearest neighbor network with 1 cycle latency, a 1-bit reconfigurable broadcast network with 10 cycle latency, a 3 cycle OR feedback circuit, and a 20 cycle count feedback circuit.

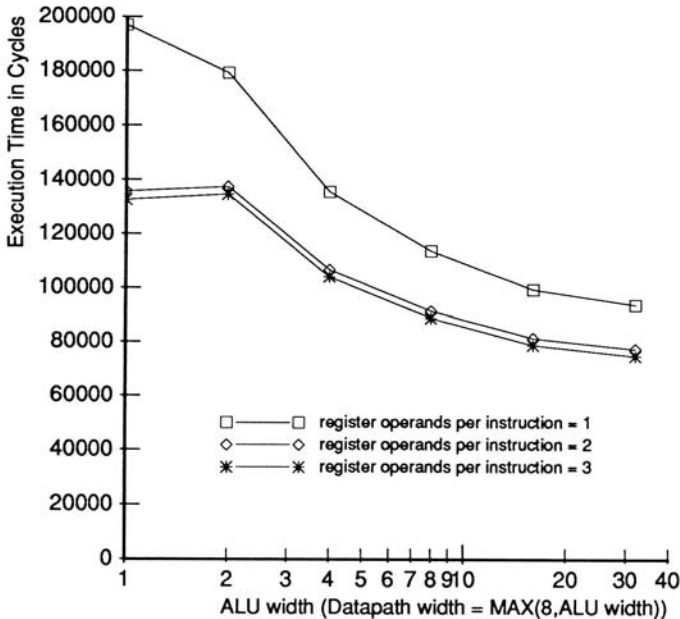


Fig. 7. Plot of the execution time of the test program described in Table 4 as a function of ALU size and number of register operands per instruction cycle.

The sample results are presented in Fig. 7. Note that most of the gain comes from increasing the ALU width from 1 to 8. This is because most of the operations are on bit, char, and short int Plane types. Also note that little gain comes from increasing the number of registers per instruction from 2 to 3. This is because the code is optimized to get rid of 3-operand instructions whenever possible and because of the nature of the computation, this can usually be accomplished.

Also significant is that there is only a factor of 2.6 improvement from minimum to maximum performance. If the datapath is *decreased* from 8 to 1 in the case of the 1 bit ALU, the factor is still only 2.95. This is because there are several virtual machine instructions whose execution times are not decreased significantly by *any* changes in the ALU. The proportion of the total execution time of these instructions therefore increases in relation to the decrease of the total execution time. For example, inter-PE communication instructions go from 13% of execution time when the ALU has width 1 and supports a single register reference per cycle to 33% when the ALU has width 32 and supports 3 register references per cycle. Similarly, the proportion of feedback instruction time goes from 2% to 5%, bit operations such as activity bit manipulation from 7% to 19%, and memory reads and writes from 4% to 9%.

5.2. Evaluating the Memory Hierarchy

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.

The critical metric in evaluating the register architecture is the number of load and store instructions required to execute a given program. The register performance depends on the number and type of registers. The critical metric in evaluating the cache performance is the hit rate; the important cache parameters are the cache size, block size, and associativity. Other parameters in the evaluation of the memory hierarchy are the number of cache levels (and parameters for each), and the bandwidth and latency between levels.

The basic problem in evaluating a potential memory hierarchy design in our system is that the virtual machine has a flat memory space with locations specified only by variable name and type: the execution traces do not reference physical memory and register locations. This is a side effect of being able to evaluate the traces with respect to any number of target machine register/cache designs without having to rerun the simulations. As a consequence, the physical memory, cache, and register behavior of the program execution must be (re)constructed *a posteriori*.

The performance is derived by executing a series of trace transformations to extract information implicitly contained therein. At a very high level, the general flow is as follows:

- Virtual machine tags are determined as being either static, dynamic, or temporary variables.
- Expressions are extracted and optimized.
- The variables are assigned virtual memory addresses.
- Registers are allocated and loads and stores inserted into the trace.
- The load/store trace is used for cache simulation using standard techniques.²⁸

For our register file evaluation example, we use the same test program as above. However, since there were few memory references in the original version (the working set is relatively small), we have this time simulated a processor that is one fourth the size of the images being processed so that each physical PE must simulate 4 virtual PEs. The effect of varying the register file size on the number of memory references is shown in Figs. 8 and 9. The two knees shown in the different figures probably correspond to working set sizes of different parts of the test program.

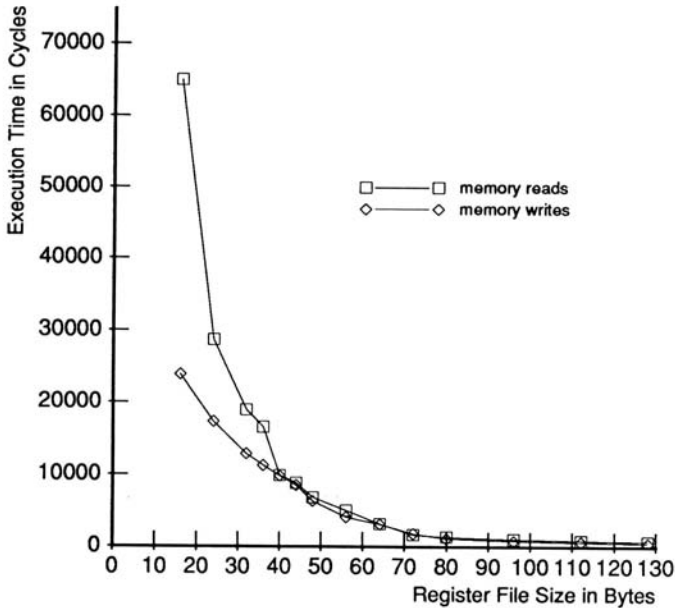


Fig. 8. Plot of the execution time of the memory reads and writes as a function of register file size in bytes. The application is the test program described in Table 4 runs with a virtualization factor of 4, i.e. each physical PE simulates 4 virtual PEs. Note knee in curve at register file size of 40.

5.3. Evaluating the Communication Network

Those inter-PE communication operations with data independent performance — for most target architectures this includes nearest-neighbor moves — can be evaluated in the same way as datapath operations. In dedicated routing networks, however, the communication performance is often data dependent.^{29,33} For these networks, detailed simulation is necessary. We have written parameterized simulators both for self-routing circuit-switched and for combining packet-switched networks. These subsume the MP1 and CM-2 routing networks, respectively. The network simulation parameters are given in Tables 5 and 6.

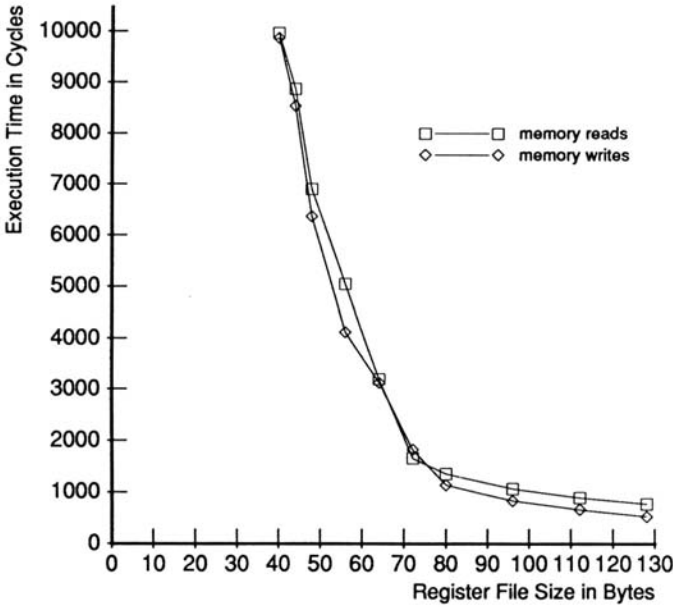


Fig. 9. Plot of the execution time of the memory reads and writes as a function of register file size in bytes. The application is the test program described in Table 4 run with a virtualization factor of 4, i.e. each physical PE simulates 4 virtual PEs. Note second knee at register file size of 80.

Table 5. Options and their parameters for the circuit-switched router simulator.

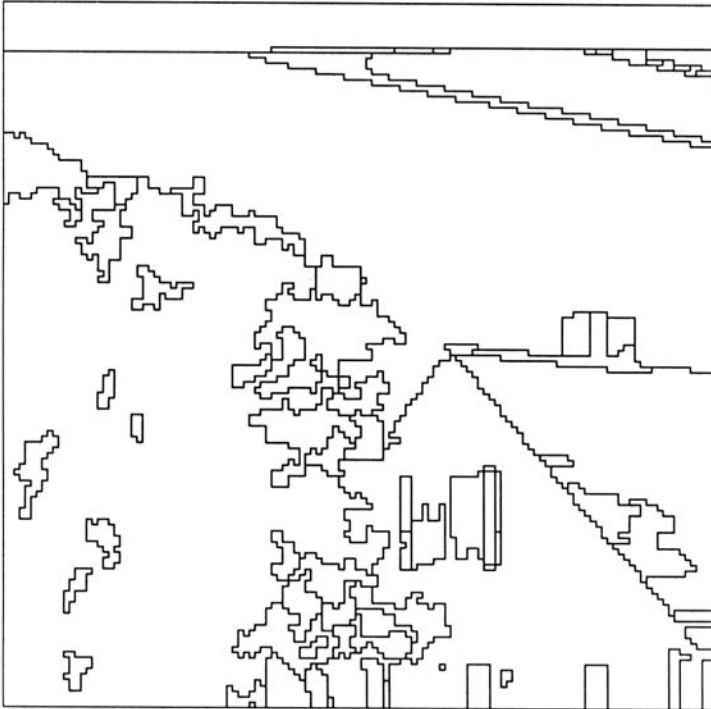
Option	Parameters
Topology	Butterfly, Omega, Baseline
Switch size in each level	$2 \times 2, \dots, n \times n$
Number of switches per level	$1, \dots, n/2$
Redundancy per output port per level	$1, 2, \dots, (\text{switch size})^2$
Number of levels	$1, \dots, \log(n)$
Latency per switch	time
Latency per wire	time
Startup latency	time
PEs per router port	$1, \dots, n$

One problem with this scheme is that data dependent communication has a very large context, on the order of a megabyte per operation. This amount of information is far too great to carry along as part of the virtual machine execution trace. Instead, we perform the router simulations during the initial program execution. Although this does slow down the virtual machine execution, the cost of simulating these instructions (e.g. global permutations) is critical only when they also dominate the cost of program execution on the target machine itself.

Table 6. Options and their parameters for the packet-switched router simulator.

Option	Parameters
Number of dimensions	$1, \dots, \log(n)$
Path width	$1, \dots, 64$
Queue size	$1, \dots,$
Latency per switch	time
Latency per wire	time
Startup latency	time
PEs per router port	$1, \dots, n$

We demonstrate the use of the packet routing simulator on a region segmentation application. A critical routing pattern that is used during the region characterization phase of a region merging algorithm³ is for every PE in a region to send data to a unique destination within that region. This facilitates the computation of region parameters such as size, border length, and the moments of various spectral values. The label images we wish to characterize are of the type shown in Fig. 10.

Fig. 10. Segmentation of a 128×128 subimage of a house scene.

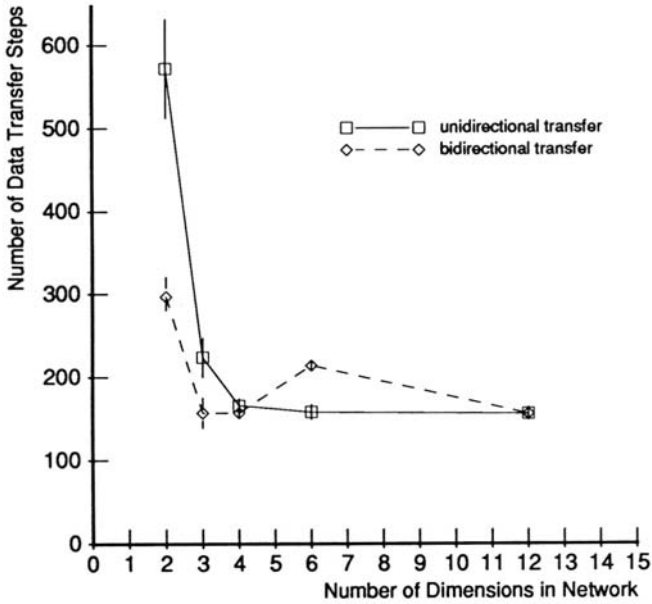


Fig. 11. Plot of the number of data transfer steps required to perform region reductions of image shown as a function of the number of dimensions of a 4096 node k -ary n -cube routing network.

The number of packet transfer steps was measured as a function of the number of dimensions in the network. For example, a k -ary n -cube with 4096 nodes can have the (k,n) pairs (2,64), (3,16), (4,8), (6,4), or (12,2). The last of these is a 12-dimensional hypercube. The results are presented in Fig. 11. Observe that, at least for these routing patterns, the utility of using more than four dimensions is questionable. Also observe that for two dimensional networks, the capability of using wires to transfer packets in both directions (on alternating cycles) results in a factor of two speed-up.

6. CASE STUDY: MEMORY HIERARCHY AND VIRTUALIZATION

In this section, we show the effect on performance of processor virtualization (the simulation of multiple virtual PEs by a single physical PE) on a CAAPP-like target machine. We again use the bottom-up and intermediate portions of the DARPA IU Benchmark as our test program.

The graph in Fig. 12 shows the effect of virtual processor emulation on the execution time. There are two things to notice. The first is the nearly linear slowdown (over the ideal behavior) that occurs when the virtualization factor 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 virtualization factor goes to 16. The reason for this can be seen in Fig. 13: for smaller virtualizations, 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.

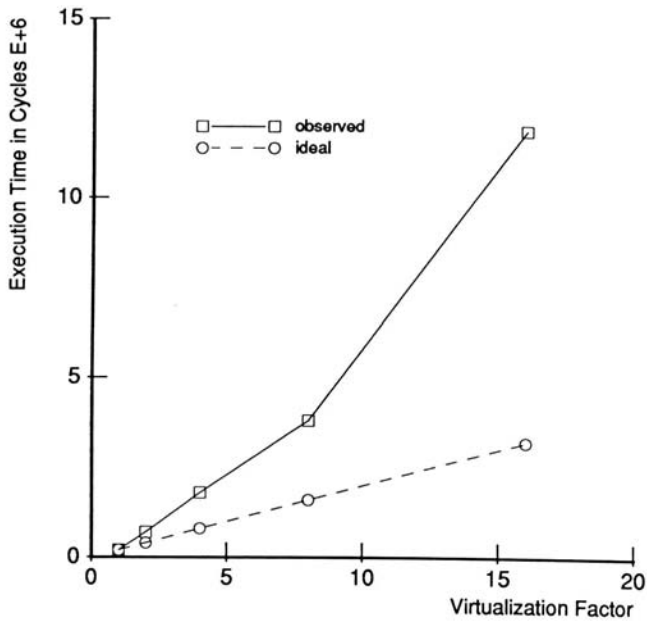


Fig. 12. Plot of the execution time of the test program described in Table 4 as a function of the PE virtualization factor. The register file size is 100 in each case. The second plot depicts behavior that would occur if there were no slowdown due to virtual processor emulation.

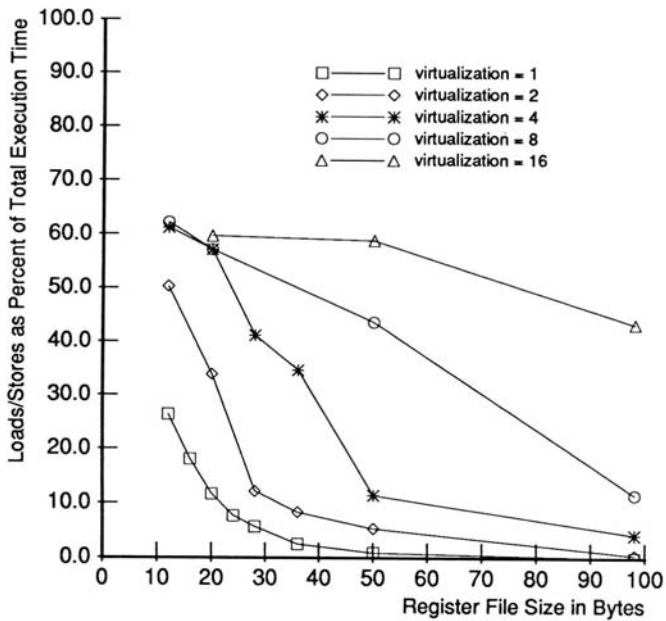


Fig. 13. Plot of the percentage of the total execution time spent on memory access as a function of the size of the register file for different virtualization factors. Test program is described in Table 4.

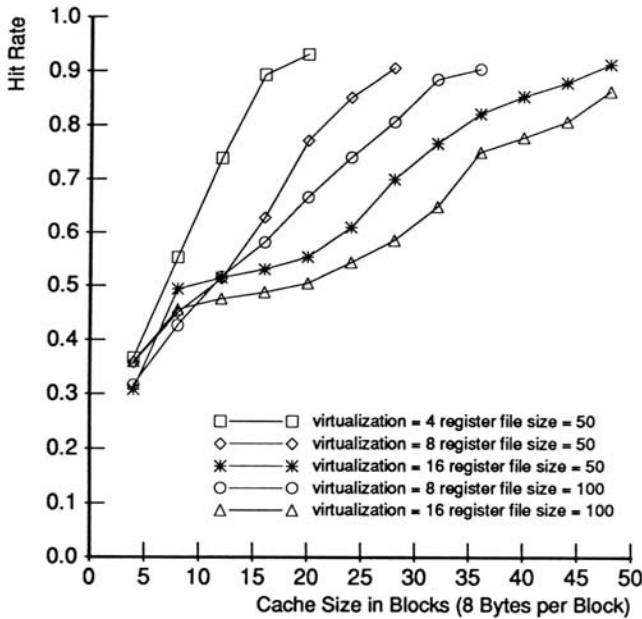


Fig. 14. Plot of the hit rate versus the cache size in 8 byte blocks. The cache is fully associative.

The practical result given in Fig. 13 is the minimum register file size for different virtualization factors so that memory fetches do not 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 for larger virtualization factors very little of the context remains after all of the virtual PEs in a code block have been emulated. Since the register file size cannot be increased indefinitely, the architectural answer is to interpose a level of cache into the design.

Figure 14 contains the hit rates of caches of various sizes on the memory reference traces generated for Fig. 13. 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 as 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 Fig. 14 is that a cache size of 400 bytes per PE suffices to achieve a 90% hit rate even for a virtualization factor of 16. Figure 15 shows how balance is achieved when a cache of that size is added to the design.

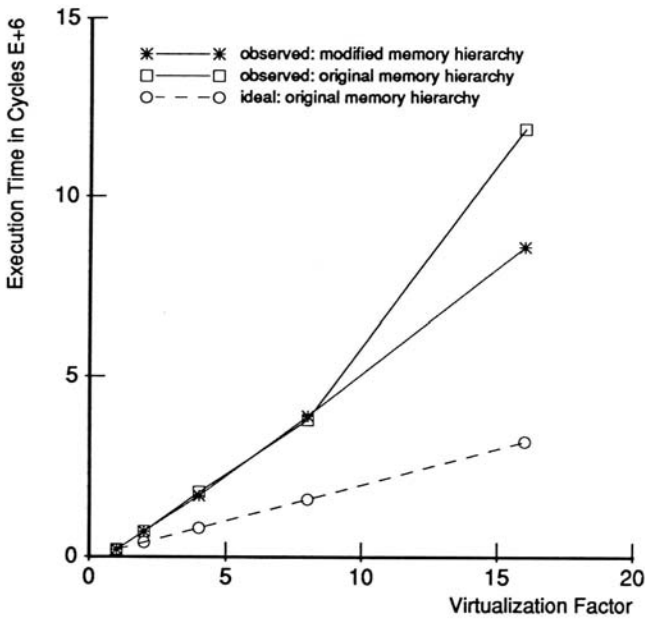


Fig. 15. Plot of the execution time of the test program described in Table 4 as a function of the PE virtualization factor. The register file size this time has been decreased to 50, but a 400 byte cache has been added. Graphs from an earlier figure are included for comparison.

7. CONCLUSION

In this article we have presented a software system architecture for use in evaluating massively parallel arrays for spatially mapped computation. This system enables us to extend previous evaluation efforts by simultaneously addressing the issues of flexibility of the design space, efficiency of the simulations, programmability of the test suite, and fairness to all target architectures within our domain, all while maintaining a high level of accuracy. In particular:

- In order to achieve maximum flexibility in evaluating the design space while still allowing efficient simulations, we have combined trace-driven simulation methodology with the emulation of the generic MPA presented by the ICL virtual machine model.
- We have addressed the problem of programmability of the test suite while maintaining comparable efficiency on all target architectures in our design space by using operator and application function libraries. The first is general purpose, consisting of emulations of useful parallel constructs. The second is application specific and contains different versions of critical sub-tasks.
- We are modeling the workload with a series of application tasks used in a real computer vision research environment.

Another interesting result is the transformation process that is performed on the virtual machine traces to reconstruct *a posteriori* the compiler optimizations,

register allocation, and caching behavior. This is an essential component in being able to run programs on a generic MPA emulator and yet tell what would have taken place had the program been run on any given target architecture. Finally, we have demonstrated the usefulness of our system on the critical problem of assessing memory architectures with respect to varying factors of processor virtualization.

REFERENCES

1. T. Axelrod, P. Dubois, and P. Eltgroth, "A simulator for MIMD performance prediction: Application to the S-1 MkIIa multiprocessor", *Parallel Comput.* **1** (1984) 237-274.
2. K. E. Batcher, "Design of the massively parallel processor", *IEEE Trans. Computers* **C-29** (1980) 836-840.
3. J. R. Beveridge, J. Griffith, R. R. Kohler, A. R. Hanson, and E. M. Riseman, "Segmenting images using localized histograms and region merging", *Int. J. Computer Vision* **2** (1989) 311-347.
4. T. Blank, "The MasPar MP-1 architecture", *Proc. 35th IEEE Computer Society Int. Conf.*, 1990, pp. 20-24.
5. D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif, "BLITZEN: A highly integrated massively parallel machine", *J. Parallel Distri. Comput.* **8** (1990) 150-160.
6. M. Boldt, R. Weiss, and E. M. Riseman, "Token-based extraction of straight lines", *IEEE Trans. Syst. Man Cybern.* **19** (1989) 1581-1594.
7. J. B. Burns, A. R. Hanson, and E. M. Riseman, "Extracting straight lines", *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-8** (1986) 425-455.
8. J. H. Burrill, *The Class Library for the IUA Tutorial*, Amerinex Artificial Intelligence, Amherst, MA, 1992.
9. D. W. Clark, "Cache performance in the VAX-11/780", *ACM Trans. Computer Syst.* **1** (1983) 24-37.
10. R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice parallel processing testbed", *Proc. 1988 ACM Conference on Measurement and Modeling of Computer Systems*, 1988, pp. 4-11.
11. P. -E. Danielsson and T. S. Ericsson, "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.
12. R. L. Davis, "The ILLIAC IV processing element", *IEEE Trans. Computers* **C-18** (1969) 800-816.
13. J. J. Dongarra, "Performance of various computers using standard linear equations software", Rep. CS-89-05, Computer Science Department, University of Tennessee, 1989.
14. M. J. B. Duff, "Review of the CLIP image processing system", *Proc. National Computing Conf.*, 1978, pp. 1055-1060.
15. R. Dutta and C. C. Weems, "Parallel dense depth from motion on the image understanding architecture", *Proc. 1993 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition*, 1993, pp. 154-159.
16. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, San Mateo, CA, 1990.
17. M. C. Herbordt, J. C. Corbett, C. C. Weems, and J. Spalding, "Practical algorithms for online routing on fixed and reconfigurable meshes", *J. Parallel Distri. Comput.* **20** (1994) 105-125.
18. M. C. Herbordt, *Evaluation of Massively Parallel Array Architectures*, Ph.D. dissertation, Department of Computer Science, University of Massachusetts, 1994.

19. M. C. Herbordt, C. C. Weems, and M. J. Scudder, "Non-uniform region processing on SIMD arrays using the coterie network", *Mach. Vision Appl.* **5** (1992) 105-125.
20. P. P. Jonker, *Morphological Image Processing: Architecture and VLSI Design*, Kluwer, The Netherlands, 1992.
21. E. R. Komen, *Low-Level Image Processing Architectures: Compared for Some Nonlinear Recursive Neighborhood Operations*, Ph.D. dissertation, Technical University Delft, The Netherlands, 1992.
22. D. E. Lang, T. E. Agerwala, and K. M. Chandy, "A modeling approach and design tool for pipelined central processors", *Proc. 6th Int. Symp. Computer Architecture*, 1979, pp. 122-129.
23. W. B. Ligon III and U. Ramachandran, "An empirical methodology for exploring reconfigurable architectures", *J. Parallel Distrib. Comput.* **19** (1993) 323-337.
24. W. B. Ligon III, *An Empirical Evaluation of Architectural Reconfigurability*, Ph.D. dissertation, Department of Computer Science, Georgia Institute of Technology, 1992.
25. J. J. Little, G. E. Bletloch, and T. A. Cass, "Algorithmic techniques for computer vision on a fine-grained parallel machine", *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-11** (1989) 244-257.
26. T. C. Marek and E. W. Davis, "Quantitative studies of processing element granularity", *Proc. 4th Symp. on the Frontiers of Massively Parallel Computation*, 1992, pp. 551-552.
27. T. C. Marek, *Comparative Evaluation of Processor Architectures for Massively Parallel Systems*, Ph.D. dissertation, Department of Electrical and Computer Engineering, North Carolina State University, 1992.
28. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies", *IBM Syst. J.* **9** (1970) 78-117.
29. D. W. Myers and G. B. Adams II, "Benchmarking and performance analysis of the CM-2", TR 88.19, NASA Ames Research Center, 1988.
30. J. R. Nickolls, "The design of the MasPar MP-1: A cost effective massively parallel computer", *Proc. 35th IEEE Computer Society Int. Conf.*, 1990, pp. 25-28.
31. K. J. Overton and T. E. Weymouth, "A noise reducing preprocessing algorithm", *Proc. Pattern Recognition and Image Processing*, 1979, pp. 498-507.
32. D. Parkinson and C. R. Jesshope, "The AMT DAP 500", *Proc. 33rd IEEE Computer Society Conf.*, 1988.
33. L. Prechelt, "Measurements of MasPar MP-1216A communication operations", Tech. Rep. 01/93, Fakultät für Informatik, Universität Karlsruhe, 1993.
34. K. Preston, "The Abington cross benchmark survey", *IEEE Computer* **22** (1989) 9-18.
35. S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The Wisconsin wind tunnel: Virtual prototyping of parallel computers", *Proc. 1993 ACM Conf. on Measurement and Modeling of Computer Systems*, 1993, pp. 48-60.
36. A. Rosenfeld, "A report on the DARPA image understanding architectures workshop", *Proc. Image Understanding Workshop*, 1987, pp. 298-301.
37. A. J. Smith, "Cache memories", *Comput. Surv.* **14** (1982) 473-530.
38. L. Snyder, "Type architectures, shared memory, and the corollary of modest potential", *Ann. Rev. Computer Sci.* **1** (1986) 289-317.
39. *Systems Performance Evaluation Cooperative*, *SPEC Newsletter: Benchmark Results*, Waterside Associates, Fremont, CA, 1990.
40. *Connection Machine Model CM-2 Technical Summary*, Tech. Rep. HA87-4, Thinking Machines Corporation, 1987.
41. *C* Programming Guide*, Thinking Machines Corporation, Cambridge, MA, 1990.
42. J. K. Tsotsos, "A 'complexity level' analysis of immediate vision", *Int. J. Computer Vision* **1** (1988) 303-320.

43. S. H. Unger, "A computer-oriented toward spatial problems", *Proc. IRE* **47** (1959) 1744-1750.
44. C. C. Weems, *Image Processing on a Content Addressable Array Parallel Processor*, COINS TR 84-14 and Ph.D. dissertation, Department of Computer Science, University of Massachusetts, 1984.
45. C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, J. G. Nash, and D. B. Shu, "The image understanding architecture", *Int. J. Computer Vision* **2** (1989) 251-282.
46. C. C. Weems, "Architectural requirements of image understanding with respect to parallel processing", *Proc. IEEE* **79** (1991) 537-547.
47. C. C. Weems, E. M. Riseman, A. R. Hanson, and A. Rosenfeld, "The DARPA image understanding benchmark for parallel computers", *J. Parallel Distrib. Comput.* **11** (1991) 1-24.

Received 15 November 1993; revised 26 July 1994.
