An Environment for Evaluating Architectures for Spatially Mapped Computation: System Architecture and Preliminary Results [*][†]

Martin C. Herbordt[‡]   Charles C. Weems

Department of Computer Science, University of Massachusetts

Abstract: An environment which addresses several problems in evaluating massively parallel array architectures is described. 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 by combining virtual machine emulation with trace driven simulation. The trade-off between fairness to diverse target architectures and programmability of the test programs is addressed through the use of operator and application libraries. Initial results are presented indicating the appropriate balance between register file and cache to optimize performance under varying levels of processor element virtualization.

[*]This paper also appears in the Proceedings of Computer Architectures for Machine Perception '93.

[†]Authors' address: Department of Computer Science; University of Massachusetts; Amherst, MA 01003; NetAd : {herbordt,weems}@cs.umass.EDU. 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).

[‡]M.C. Herbordt is supported in part by an IBM Fellowship.

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 [23]. The only way such computation rates are physically possible is by using massively parallel processors. The question this research addresses is how to go about making architectural decisions for these types of machines on the subset of vision computations we refer to as spatially mapped computations.[1] In particular, we describe a methodology, software system architecture, and preliminary results in the use of trace analysis of real program executions for the architectural evaluation of massively parallel processor arrays.

Most previous vision architecture studies have been based on either mapping sample algorithms to architectures (e.g. [5]), requirements analysis (e.g. [20]), or feedback from benchmarks (e.g. [21]). The first two of these methods have served their purpose in making 'first passes' at machine architectures, but now need to be extended to yield more specific and detailed results. Benchmarking efforts have also had their drawbacks:

1. **Appropriate Workload.** The benchmark test suites may not have accurately reflected the workload of a vision system. They have often been restricted to relatively small computations and a set of well-known—but not necessarily representative—algorithms. Recent efforts have gone a some way in changing this [24].

2. **Flexibility versus Efficiency and Accuracy.** The classic choice in architectural evaluation is between extensive analysis of existing hardware (or of a detailed design) and a usually less accurate analysis of a parameterized model. Previous benchmarks have been of the former kind and thus have neither explored a large part of the ar-

[1]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.

chitectural design space, nor investigated the effects of varying multiple parameters simultaneously.

3. **Fairness versus Programmability.** This is the problem of not allowing a test suite to include codes that are inherently unsuitable for certain target architectures, while maintaining the ability to port code to new designs with minimal effort. Benchmarks have leaned towards the fairness side by being task oriented [16, 17, 24] 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 specific machines (or their designs).

We address the first problem by including applications in our test suite that are in continual use in a machine vision research environment. These programs have significant size and established utility. Flexibility while maintaining enough efficiency to explore a signficant part of the design space is achieved by combining virtual machine emulation—that is, behavioral simulation that generates traces of virtual machine code—with trace driven simulation. The first part is orders of magnitude faster than detailed simulation, while the second part allows for 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 specification and handcoding. 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.

There are two primary results in this paper. The first is the software system architecture for an evaluation environment that is flexible, efficient, fair, and programmable. The second is experimental data with respect to memory hierarchy and virtual processor emulation. The latter result is significant in that it demonstrates the efficacy of our approach in evaluating detailed designs, and provides recommendations for architectural decisions for future generation array processors.

2 The Architecture and Application Domains

2.1 The Architectural Design Space

The design space has two sets of components: the first is common to all the architectures under investigation, while the second consists of those components that will be varied and evaluated. We describe the common part first.

The architectures are all massively parallel arrays (MPAs) with 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. The distinction was inspired by Snyder's work [18] and is defined formally in [9]. Roughly speaking, a *parameter* is a component for which a reasonable compiler could be expected to make algorithmic decisions without user input; the opposite is true of *features.* In general, all 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 the presence or absence of which 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 features. This is because, for several tasks (e.g. connected components, convex hull), these networks will have different optimal algorithms. See Figure 1 for an example of how various popular MPAs can be viewed as

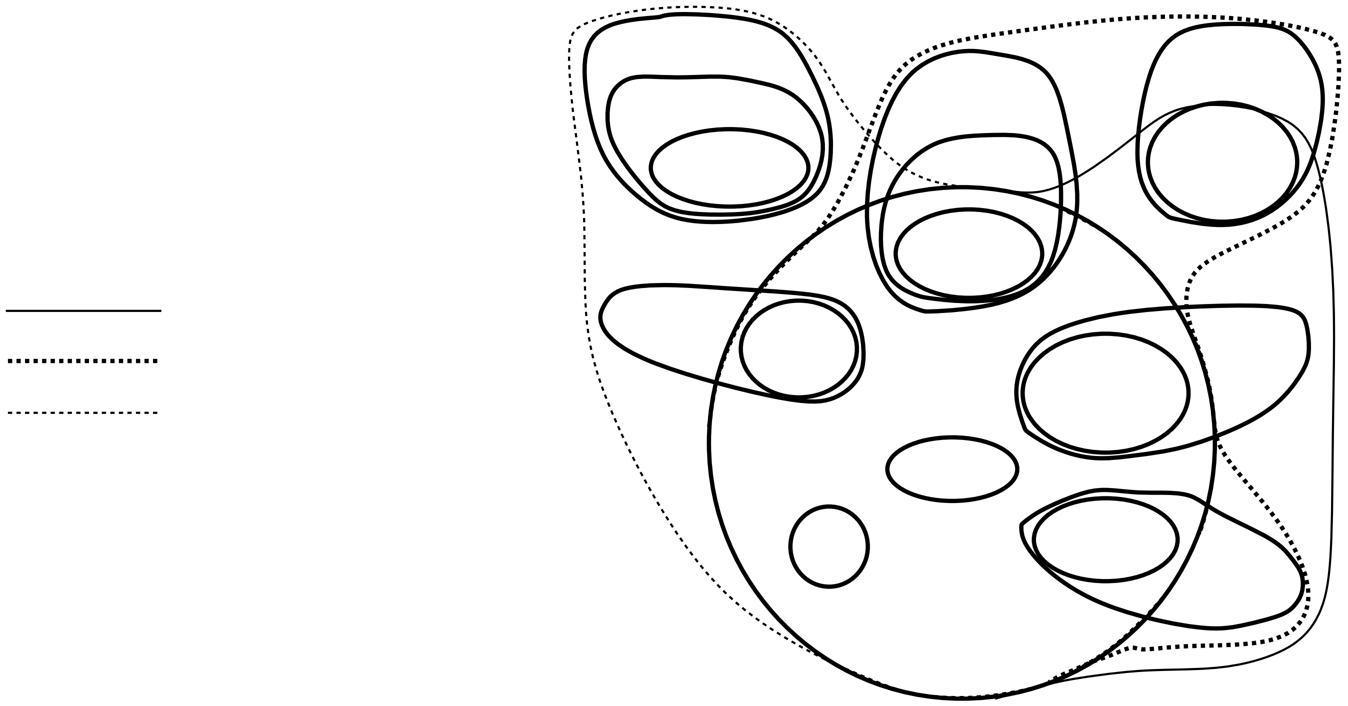

Figure 1: Representation of three massively parallel arrays (MPAs) as collections of core and optional architectural features.

Application	Description and Comments
DARPA IU Benchmark II	Synthetic recognition task developed to evaluate complete image understanding systems [24]. Uses the low-level bottom-up and the intermediate-level processing components.
Weymouth-Overton Preprocessor	Information preserving image filter. Uses edge-model curve fitting [14].
Fast Line Finder	Based on Burns' algorithm [3]: segments image by gradient orientation and fits line to resulting regions.
Depth From Motion	Computation dominated by correspondence-based matching [7].
Boldt's Line Finder	Perceptual organization based. Iterative grouping algorithm [2].
Region Segmentation	Based on Nagin-Kohler system [1]. Combines histogram-based image splitting and region merging techniques.

Table 1: Description of test suite programs

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	corresponence 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

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

collections of features.

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 had two criteria in selecting the tasks.

The first was to make sure that the tasks, as a suite, contained a representative sample of most of the *types* of computations found in the domain of spatially mapped computation. This was done to ensure that designs are excercised fully: even if the weight of a particular type of computation does not match that in a true workload, no significant architectural weakness should go undetected.

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

3 The Virtual Machine Evaluation Methodology

In this section we describe how we address the flexibility/efficiency issue in architectural evaluation. In particular, we did not want to use a detailed simulator as that would be much too slow to ever search more than a very small amount of the design space. Neither did we want to lose much accuracy.

The basic idea is to use the principle of trace driven simulation: run a program on an emulator, generate a trace, and analyze the trace with respect to architectural models to derive performance results. Where our implementation differs from the common usage of this technique is that the emulator does not need to resemble the target architecture in any respect: it only needs to be able to run C++.

The test suite programs written in ICL are compiled into virtual machine code and executed on a virtual machine simulator. The virtual machine model is that of the generic MPA presented by the ICL programming language [4]. ICL is a C++ extension with similar semantics to other data parallel programming languages. It is like C* [19], 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 several benefits.

- The test suite programs can be executed efficiently and traces generated on any platform that supports C++. The test suite code is run with similar efficiency as code written especially for a serial processor. While this still yields the inherent slowdown from simulating to simulated machines (perhaps a factor of 100-1000), it is approximately 100 times faster than running the ICL code on a detailed MPA simulator.
- Since the program traces are evaluated off-line, they can be generated much less frequently, and only once to test any combination of components in the memory hierarchy, PE internals, and most communication operations with data-independent performance.

The trace is analyzed to derive performance with respect to the three primary architectural components: memory hierarchy, internal PE datapath, and interPE communication, as described below.

3.1 Evaluating the Datapath

A generic MPA datapath (i.e. PE ALU and other internals) has been specified and a parameterized microcode generator written for it. The microcode generator takes as input the datapath design parameters and outputs the corresponding microcode for any of the possible virtual-machine-instruction/operand-type combinations. See Table 3 for a list of parameters. For simple ALUs, the time complexity of each virtual machine instruction is then computed and tabulated and the virtual machine instruction trace evaluated by a series of table lookups. For more complex ALUs, e.g. those with pipelining, further processing is needed to determine the gaps that occur when the pipe is flushed or on memory delays. In this case, the memory evaluation processing must be completed first to integrate the off-chip memory references into the instruction trace.

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

3.2 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 effects of compiler optimization and controller support for dynamic allocation are also addressed, but will be discussed elsewhere.

The basic problem in evaluating a potential memory hierarchy design is that the virtual machine has a flat memory space with locations specified only by variable names and types: 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. It does mean, however, that 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.
- 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 [12].

3.3 Evaluating the Communication Network

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 [13, 15]. For these, detailed simulation is necessary. We have written parameterized simulators for both self-routing circuit-switched and combining packet-switched networks. These subsume the MP1 and CM-2 routing networks, respectively. The network simulation parameters are given in Table 4 and Table 5.

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

Table 4: Options and their parameters for the circuit-switched router simulator.

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

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

One problem with this scheme is that data dependent communication has a very large context, on the order of a mega-byte per operation. This amount of information is far too great to carry along as part of the virtual machine execution trace. What we have done instead is 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 significant only when they also dominate the cost of program execution on the target machine itself.

4 System Architecture

In this section we discuss our overall system architecture and address the issue of fairness while retaining programmability.

Architects of general purpose processors have established the benefits of using trace driven simulation for architectural evaluation [8]. 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 SPEC [19] and LINPACK [6] 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 the 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 orientation of most

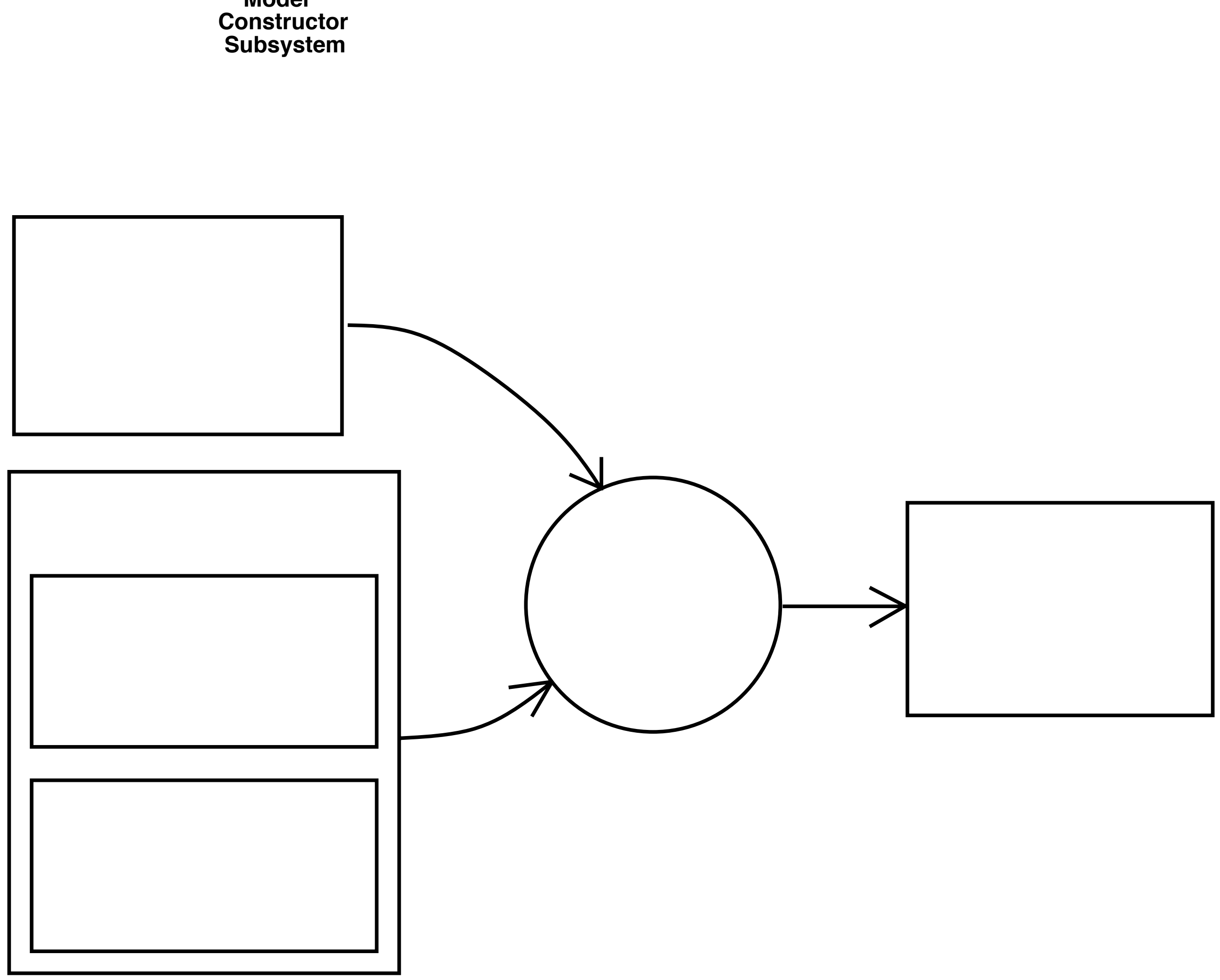

Figure 2: ENPASSANT system architecture: highest level view.

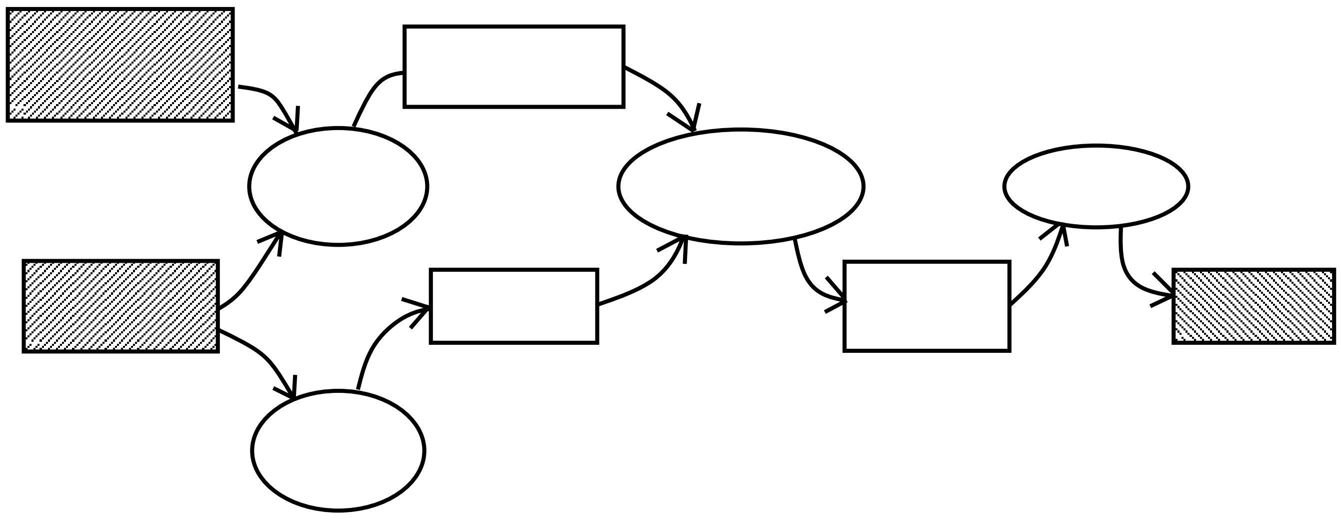

Figure 3: ENPASSANT system architecture: block diagram.

parallel vision benchmarks) and address it as follows: sections of the test programs that require different algorithms for efficiency reasons will have them provided. This turns out to be a reasonably small number for reasons that will be discussed below.

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 2). 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 3).

- The **input constructor** takes as input application programs written in ICL and outputs code executable by the virtual machine simulator.
- The **virtual machine simulator** runs the virtual machine code, and generates execution traces.
- The **model constructor** transforms the input architecture parameters into instruction, memory, and communication models for use by the trace analyzer.
- The **trace analyzer** inputs the virtual machine traces and the target machine models and outputs performance measures.

Much of the functionality of model constructor and trace generator and analyzer has already been described in the previous section. See Figure 4 and Figure 5 for details. The input constructor will now be described further. See Figure 6 for its components and flow.

Certain virtual machine functions are not implemented directly in hardware on all target architectures; examples are global reduction and the count responders feedback operation. The execution of these instructions must therefore be emulated by operations that *are* sup-

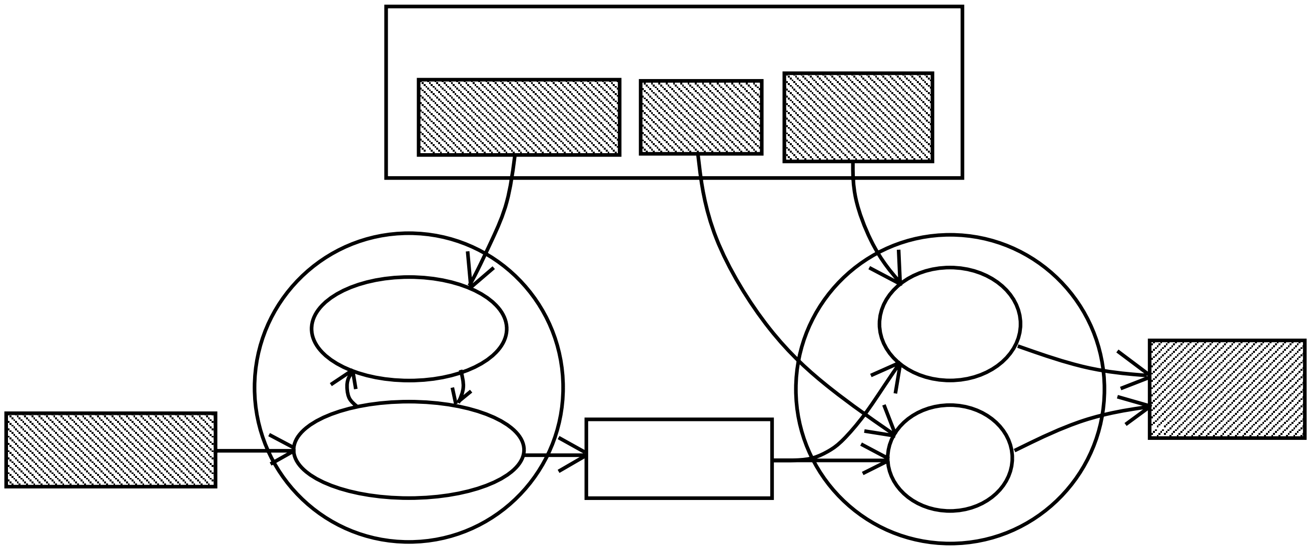
Figure 4: Trace Generation and Analysis Subsystem

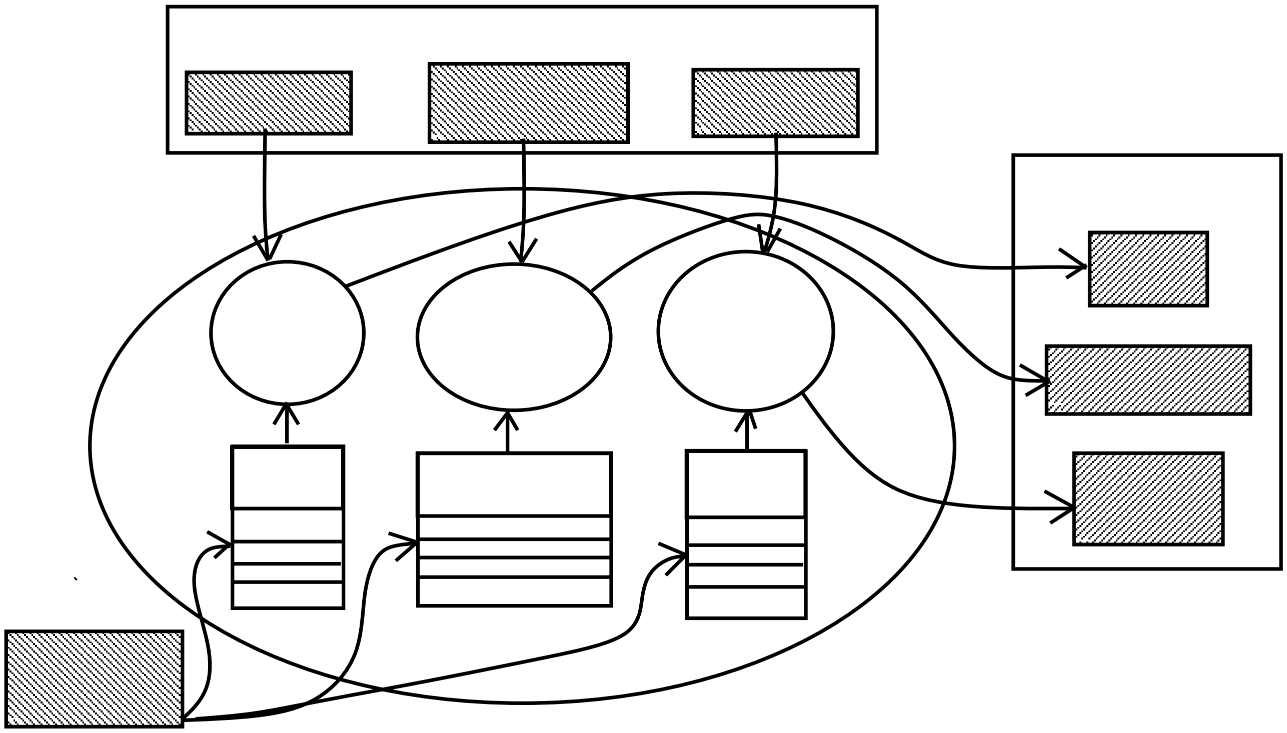
Figure 5: Model Constructor Subsystem

ported in hardware. To do this we have created the Operator Emulation Library (OEL) that emulates operators not supported on a certain target machines with functions containing operators that are.

As we discussed earlier, certain tasks have different optimal algorithms depending on the features present in the target architecture. For example, the preferred connected components labelling code on the connection machine uses either pointer jumping or segmented grid scan based algorithms [11], while that on the CAAPP uses multi-associative broadcast [10]. As we have already stated, the use of the correct algorithm is critical in making fair architectural comparisons. The Application Function Library (AFL) contains the various versions of these functions.

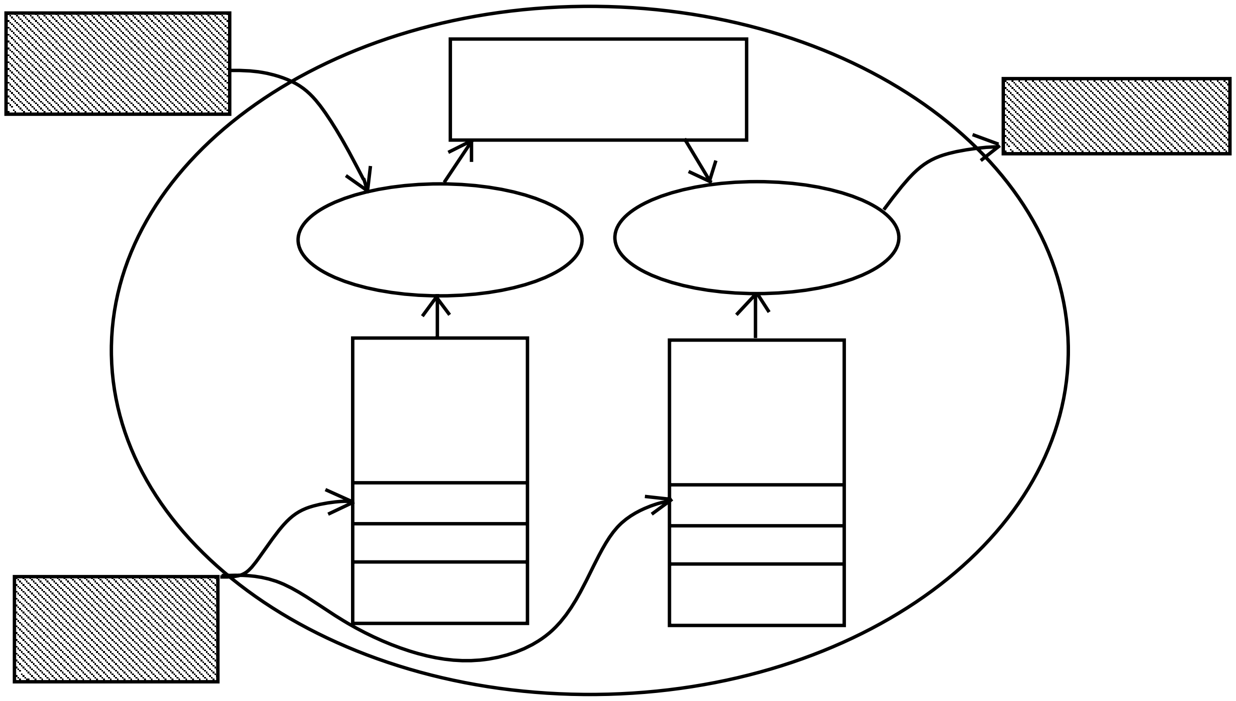

Figure 6: Input Constructor Subsystem

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 as many architectural

Low-level Processing
Label connected components
Compute the K-curvature
Extract the corners
Select components with three or more 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

Table 6: Tasks of the DARPA IU Benchmark II that are used in the test suite program.

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 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 Case Study: Memory Hierarchy and Virtualization

In this section, we show the effects of virtualization on a CAAPP-like target machine (see [22] for details) running the bottom-up and intermediate portions of the DARPA IU Benchmark. The constituent tasks can be found in Table 6.

The graph in Figure 7 shows the effect of virtualization on the execution time. There are two things to notice. The first is that the slope through the first four virtualization points is greater than 1. This can be attributed to the ever greater proportion of time needed to perform communication among virtual PEs. The second is the leap when the virtualization factor goes to 16. The reason for this can be seen in Figure 8: for smaller virtualizations, most of the working set can fit into a register file of 100 bytes per PE.

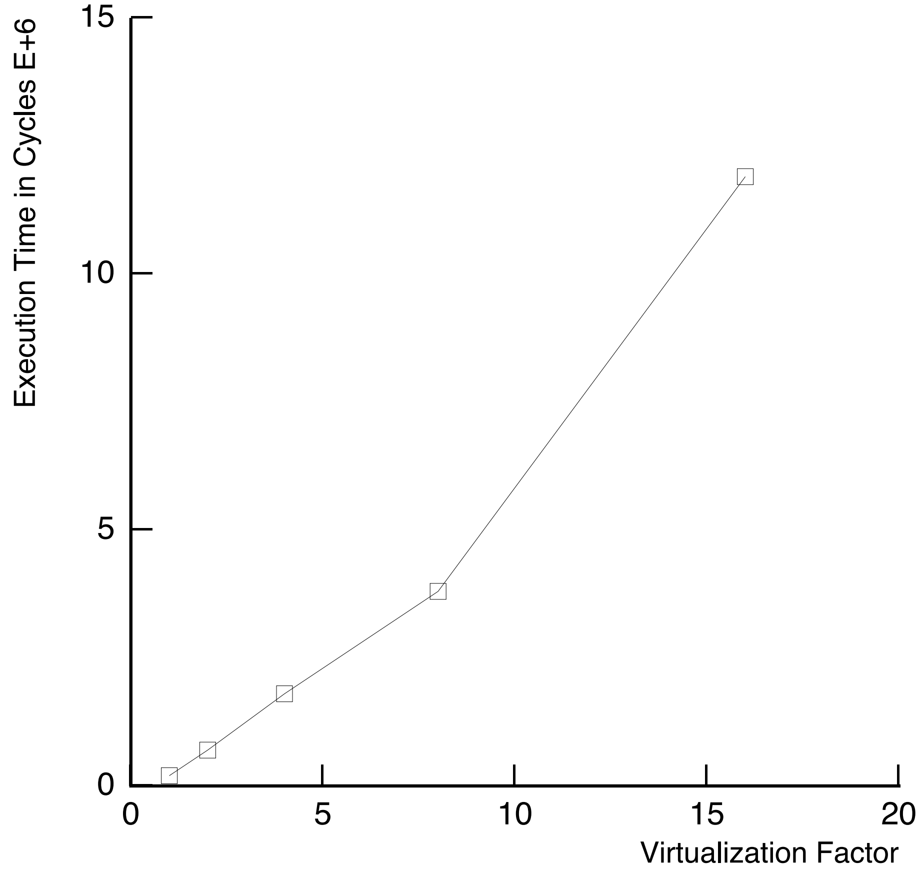

Figure 7: Plot of the execution time of the test program described in Table 6 as a function of the PE virtualization factor. The register file size is 100 in each case.

The practical result given in Figure 8 is the minimal size of the register file for different virtualization factors so that the memory fetches do not a dominate the total execution time. The main memory access time is assumed to be 5 times that of an arithmetic operation. For factor 1 and 2 virtualizations, only a relatively small register file of 25-30 bytes per PE is needed. The small size is not surprising since most of the Plane types used by the benchmark only require single byte storage. It is also apparent, however, that for larger virtualizations, very little of the context remains after all of the virtualizations of a code segment have been executed. Since the register file size cannot be increased indefinitely, the architectural answer is to interpose a level of cache into the design.

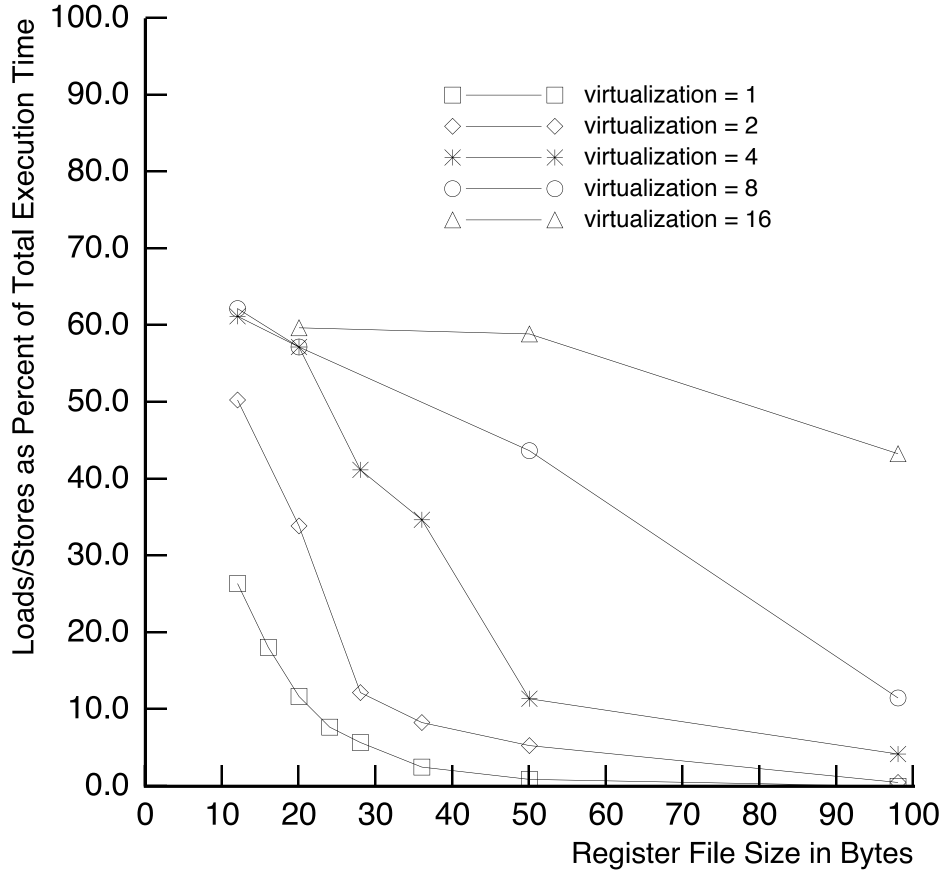

Figure 8: Plot of the percentage of the total execution time spent on memory access as a function of the size of the register file for different virtualization factors. Test program is described in Table 6.

Figure 9 contains the hit rates of caches of various sizes on the memory reference traces generated for Figure 8. 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 increased register file size reduces the absolute number of memory references, thereby decreasing the locality of those that remain.

Another result from Figure 9 is that a cache size of 400 bytes per PE suffices to achieve

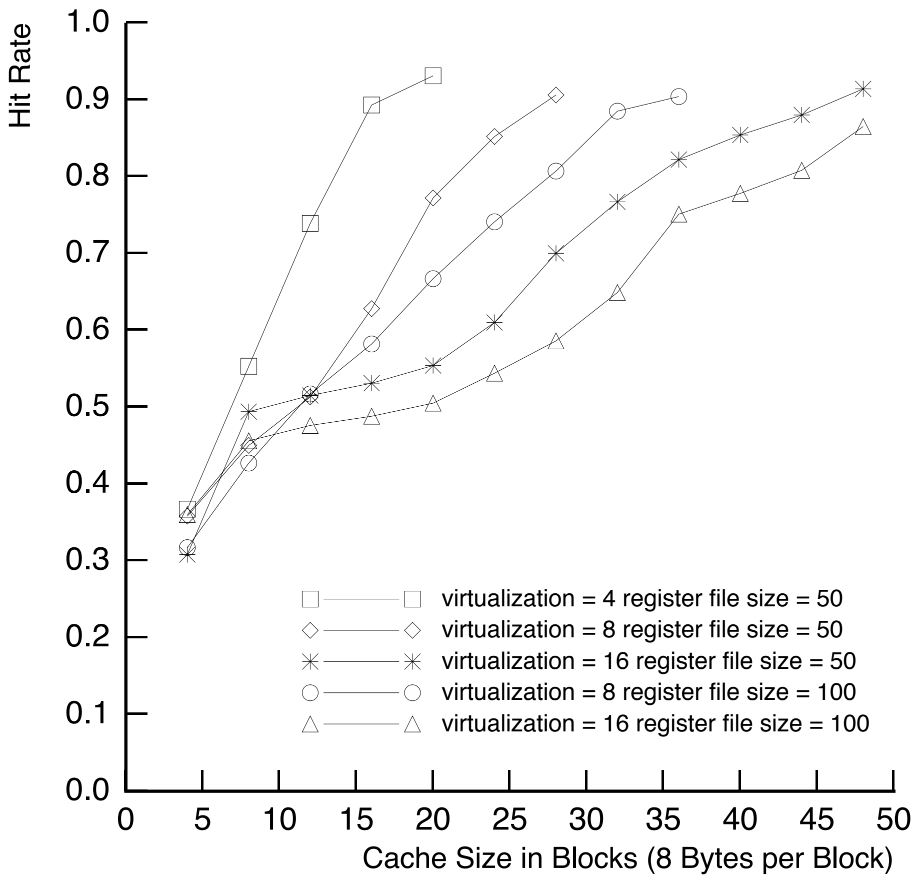

Figure 9: Plot of the hit rate versus the cache size in 8 byte blocks. The cache is fully associative.

a 90% hit rate even for a virtualization factor of 16. Figure 10 shows how balance is achieved when a cache of that size is added to the design.

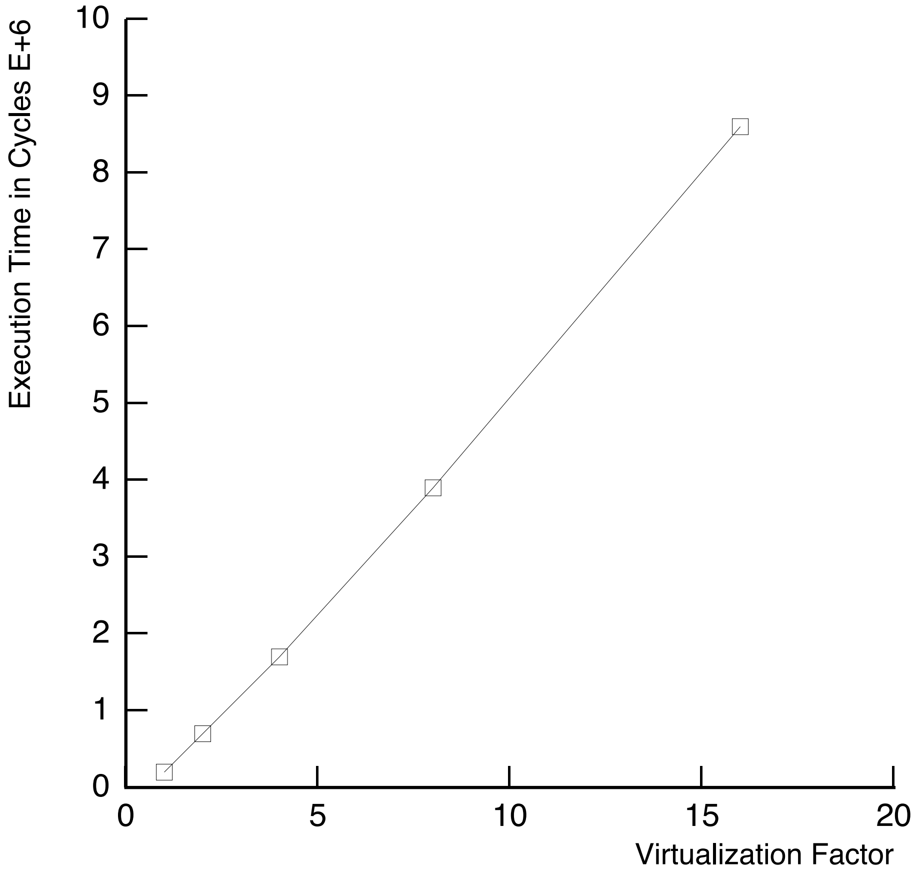

Figure 10: Plot of the execution time of the test program described in Table 6 as a function of the PE virtualization factor. The register file size this time has been decreased to 50, but a 400 byte cache has been added.

6 Conclusion

In this article we have presented a software system architecture for use in evaluating massively parallel arrays for spatially mapped computation. The system we have developed will enable 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. In particular:

- We are modeling the workload with a series of application tasks used in a real computer vision research environment.

- 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.

- 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.

Another interesting result is the transformation process that is performed on the virtual machine traces to reconstruct *a posteriori* register allocation and caching behavior. This is an essential component in being able to run programs on a generic MPA emulator and yet being able to tell what would have taken place had the program been run on any give 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] J.R. Beveridge, J. Griffith, R.R. Kohler, A.R. Hanson, E.M. Riseman (1989): "Segmenting Images Using Localized Histograms and Region Merging," *International Journal of Computer Vision,* 2 (3).

[2] M. Boldt, R. Weiss, E.M. Riseman (1989): "Token-Based Extraction of Straight Lines," *IEEE Transactions on Systems, Man, and Cybernetics,* 19 (6).

[3] J.B. Burns, A.R. Hanson, E.M. Riseman (1986): "Extracting Straight Lines," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* PAMI-8 (4) pp. 425-455.

[4] J.H. Burrill (1992): *The Class Library for the IUA Tutorial,* Amerinex Artificial Intelligence, Inc.

[5] P.-E. Danielsson and T.S. Ericsson (1983): "LIPP—Proposals for the Design of an Image Processor Array," in *Computing Structures for Image Processing,* M.J.B. Duff, editor, Academic Press, New York.

[6] J.J. Dongarra (1989): "Performance of Various Computers Using Standard Linear Equations Software," *Report CS-89-05,* Computer Science Department, University of Tennessee.

[7] R. Dutta (1993): "Depth From Motion and Stereo: Parallel and Sequential Algorithms, Robustness and Lower Bounds," *Ph.D. Dissertation; Department of Computer Science; University of Massachussetts.*

[8] J.L. Hennessy and D.A. Patterson (1990): *Computer Architecture: A Quantitative Approach,* Morgan Kaufman Publishers, Inc. San Mateo, CA.

[9] M.C. Herbordt (1992): "The Evaluation of Massively Parallel Array Architectures," *Technical Report,* Department of Computer Science, University of Massachusetts.

[10] M.C. Herbordt, C.C. Weems, M.J. Scudder (1992): "Non-Uniform Region Processing on SIMD Arrays Using the Coterie Network," *Machine Vision and Applications,* 5 (2), pp. 105-125.

[11] J.J. Little, G.E. Blelloch, T.A. Cass (1989): "Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* PAMI-11 (3), pp. 244-257.

[12] R.L. Mattson, J. Gecsei, D.R. Slutz, I.L. Traiger (1970): "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal,* 9 (2).

[13] D.W. Myers and G.B. Adams II, "Benchmarking and Performance Analysis of the CM-2," *TR 88.19,* NASA Ames Research Center.

[14] K.J. Overton and T.E. Weymouth (1979): "A Noise Reducing Preprocessing Algorithm," *Proc. Pattern Recognition and Image Processing,* pp. 498-507.

[15] L. Prechelt (1993): "Measurements of MasPar MP-1216A Communication Operations," *Technical Report 01/93,* Universität Karlsruhe.

[16] K. Preston (1989): "The Abington Cross Benchmark Survey," *IEEE Computer,* 22 (7), pp. 9-18.

[17] A. Rosenfeld (1987): "A Report on the DARPA Image Understanding Architectures Workshop," *Proceedings: Image Understanding Workshop,* 298-301.

[18] L. Snyder (1986): "Type Architectures, Shared Memory, and the Corollary of Modest Potential," *Annual Review of Computer Science,* 1, pp. 289-317.

[19] Systems Performance Evaluation Cooperative (1990): *SPEC Newsletter: Benchmark Results,* Waterside Associates, Freemont, CA.

[19] Thinking Machines Corporation (1990): *C* Programming Guide,* Thinking Machines Corporation, Cambridge, MA.

[20] J.K. Tsotsos (1988): "A 'Complexity Level' Analysis of Immediate Vision," *International Journal of Computer Vision,* 1 (4), pp. 303-320.

[21] C.C. Weems (1984): "Image Processing on a Content Addressable Array Parallel Processor," *COINS TR 84-14 and Ph.D. Dissertation,* University of Massachusetts.

[22] C.C. Weems, S.P. Levitan, A.R. Hanson, E.M. Riseman, J.G. Nash, D.B. Shu (1989): "The Image Understanding Architecture," *International Journal of Computer Vision,* 2, pp. 251-282.

[23] C.C. Weems (1991): "Architectural Requirements of Image Understanding with Respect to Parallel Processing" *Proceedings of the IEEE,* 79 (4), pp. 537-547.

[24] C.C. Weems, E.M. Riseman, A.R. Hanson, A. Rosenfeld (1991): "The DARPA Image Understanding Benchmark for Parallel Computers," *Journal of Parallel and Distributed Computing,* 11, pp. 1-24.