An Approach for Simulating an Applicative Programming Storage Architecture on the Massively Parallel Processor

by

John T. O'Donnell

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 179

An Approach for Simulating
an Applicative Programming Storage Architecture
on the Massively Parallel Processor

by

John T. O'Donnell

September, 1985

An Approach for Simulating an Applicative Programming Storage Architecture on the Massively Parallel Processor

John T. O'Donnell

Computer Science Department
101 Lindley Hall
Indiana University
Bloomington, Indiana 47405

Summary

This paper describes an Applicative Programming Storage Architecture (APSA) along with an approach for simulating it on the Goodyear Massively Parallel Processor (MPP). The objectives of this work are (1) to study the performance of APSA, and develop it further, and (2) to develop techniques and tools for simulating VLSI architectures (especially systolic architectures) on the MPP.

The APSA architecture provides the fastest known solutions for several problems that arise in the active research area of Applicative Programming Architectures — in particular, accessing variable bindings, reducing the overhead of storage management, combining list and vector data structures, unifying stack and heap implementations of continuation data structures, and solving the "aggregate update problem". Further progress on this architecture requires a fast simulator, but the simulation of VLSI designs on conventional architectures is difficult because of the enormous computational requirements.

This paper shows how the VLSI layout of the APSA architecture corresponds to the architecture of the MPP. The simulation will map each computational element (CE) of a geometric layout of the APSA onto a processor in the Array Unit of the MPP. Each Array Unit processor will contain a code specifying what type of CE it represents, as well as the local storage of that CE. The Array Control Unit will broadcast instructions to the Array Unit causing all the processors corresponding to a particular type of CE to execute simultaneously. The simulator will issue bit-level instructions to the Array Unit, exploiting the bit-serial architecture of the MPP.

1. Introduction

This paper describes a novel VLSI computer architecture (APSA) [16, 18, 19], and a method for simulating it on the Goodyear Massively Parallel Processor [20]. This work has two interesting aspects:

(1) the APSA architecture itself, which addresses many difficult problems in the active area of applicative programming architecture, and

(2) techniques for exploiting the architecture of the MPP in order to simulate VLSI architectures.

The simulation will lead to advances in both areas, and it will also provide a useful test of the MPP's capabilities for VLSI architecture simulation.

The purpose of APSA is to support fast implementation of applicative programming language interpreters. It does this by improving the time complexity of several algorithms that are critically important in such implementations. Specifically, the APSA system improves the time complexity of several key data structure operations by a factor of n, where n is the size of the data structure, and it also improves the space complexity of some algorithms. For example, [19] shows how APSA improves both the time complexity and the space complexity of the "aggregate update problem", which has received considerable attention in the research literature (*e.g.*, [7]). Section 4 briefly describes how the APSA hardware works, and Section 5 discusses its performance and compares it with the performance of conventional systems.

The remainder of this paper briefly describes the area of applicative programming architecture, the nature of the APSA architecture research and the simulation research, how they will be carried out, and the significance of the expected results.

2. Architectures for applicative programming

There is widespread agreement [3] among computer scientists that parallel execution of programs will be necessary in order to gain the performance desired for many applications — improvements in raw circuit speed will not be enough. A wide variety of ideas about how to build parallel architectures has lead to such machines as the MPP, vector processors, and MIMD networks like Cm*.

Another approach to parallelism — which leads to the APSA system — is to design architectures that exploit the parallelism inherent in applicative programs (also called functional programs). Wise has published a tutorial introduction to applicative programming [22], and there is a survey of architectures for applicative programming by Vegdahl [21]. The main premise motivating applicative architecture research [1, 6, 8, 12, 13, 17, 23] is that it is difficult to write programs that control parallelism explicitly, so we should build systems that integrate a parallel architecture with a highly effective way to program it. Many fast parallel algorithms for specific applications already exist, but applicative architecture research isn't concerned with finding the best algorithm for one particular problem; instead, it tries to bring the performance benefits of parallelism to *all* programs written in an applicative language.

The basic idea of the approach is to start with applicative or functional programming languages that encourage users to write programs that are implicitly parallel, and then to develop architectures that can automatically exploit that parallelism. Usually this does not produce the fastest imaginable implementation of an algorithm, but it can substantially improve the performance of simple, clear and correct programs.

Most applicative architectures are MIMD (multiple instruction streams, multiple data streams), and they try to identify independent expressions that can be evaluated in parallel on separate processors. It is easy to get a small amount of parallelism this way, but techniques for automatically finding massive parallelism are unknown.

The APSA system has a different philosophy. At the top level (*i.e.*, at the processor/memory/switch level [2]) it is an SISD system (single instruction stream, single data stream). Thus APSA exhibits no parallelism at all at the top level. All of APSA's parallelism occurs inside the storage system, which can execute a number of very powerful data structure operations in one clock cycle. As a result, many operations that require iteration on a conventional system require only a few cycles on the APSA system. Most applicative architectures try to use parallel processors in order to execute many instructions simultaneously, but the APSA system uses parallelism in order to reduce radically the number of instructions required by a computation. The next section explains why this is a promising approach, and briefly describes the APSA architecture.

3. Scientific rationale of the APSA

Given a program that must execute a set of instructions, there are three ways to make it run faster:

#1: use faster circuitry, cutting the time to execute each instruction,

#2: execute many instructions simultaneously on separate processors, and

#3: find a way to reduce the number of instructions that must be executed.

Faster circuitry (#1) is useful but not sufficient to meet future computation needs. Most research in high performance computer architecture (including most applicative architecture research) uses #2, parallel execution of instructions. That is also useful, but sometimes a disappointingly small amount of parallelism can be found among the instructions, and the cost of finding that parallelism may be high. The APSA system uses #3: *APSA has parallel hardware within the storage unit in order to implement powerful operations which greatly reduce the number of instructions that a program needs to execute.* An ideal high performance system would use all three methods, and would exhibit parallelism at many levels of abstraction. The APSA research is especially significant because #3 has largely been neglected.

The APSA system does not simply reduce the number of instructions executed by a constant factor. That would not be worthwhile, since there is a large constant expense factor in its construction. Instead, it performs in constant time several operations that require iteration on conventional systems. For example, some algorithms that normally require $n \log n$ time only require $\log n$ time on APSA. This accelerates a number of useful application algorithms, but the most important result is that *APSA speeds up the key algorithms in applicative language interpreters.* The speed of those algorithms is crucial, since all application programs ultimately rely on them.

The benefits of the APSA arise from the needs of data structure manipulation for applicative languages. Conventional storage units provide two "instructions" — **fetch** and **store** — and programs must implement all their data structure operations with them. Many data structures have been found that are efficiently supported by **fetch** and **store**[9], but such data structures are inadequate for the special needs of applicative programming languages. For example, accessing the values of variables in languages such as LISP [14], Scheme [5] and Daisy [10] can be very expensive because of the constraints on the representation of environments (an environment is a set of variable bindings). There are several ways to get around this problem, but none of them produce constant-time access to variables' values without undesirable tradeoffs. The real problem is that the natural representation for environments is a data structure that cannot be manipulated efficiently with just the **fetch** and **store** instructions. With its much more powerful storage instructions, the APSA can do all the necessary operations on the environment data structure in constant time.

In addition to fast access to variables, the APSA helps to solve several other problems that arise in applicative languages.

1. The APSA makes reference counting algorithms less time consuming and more effective. Reference counting enables the system to reclaim garbage and make

it available to the user again very quickly.

2. Reference counting cannot reclaim arbitrary circular structures, so "garbage collection" is occasionally necessary The APSA makes garbage collection much faster, since it is able to mark large numbers of cells simultaneously.
3. The APSA implements list structures and vector structures, and all the usual operations on lists and vectors (insertion, deletion, indexing, *etc.*) can be performed *on the same object*. Thus the APSA supports the standard data structures, but it also makes possible a variety of new data structures that have not yet been investigated since conventional computers cannot manipulate them efficiently.
4. The APSA solves the "aggregate update problem" for applicative languages. Applicative languages do not allow side effects, so programs frequently recopy data structures that may be arbitrarily large. This is extremely expensive, both in space and time. The APSA implements aggregate updates, without restrictions, using constant space and time.

4. How the APSA Hardware Works

The APSA is a storage unit that represents data structures as linear vectors, and that provides many powerful instructions for manipulating these data structures — in contrast to conventional storage units, which only provide **fetch** and **store**. The instructions that APSA implements require several basic types of computation, and the storage architecture contains data paths, combinational logic and registers to support them. Some of the requirements and corresponding architectural features are described below.

1. *storing information: cells containing register elements*
   The APSA is a storage system. The information it holds is partitioned into words, and it contains a set of "cells" that contain register hardware to store those words.
2. *linear data structures: linear organization of cells with Attached flag*
   The APSA represents linear data structures in a compact form that is similar to sequential vectors in conventional systems, and "CDR-coded lists" in LISP machines. To support this representation, the hardware has a linear organization of cells, and a special flag called *Attached* in each cell represents the "CDR codes". The *Attached* flag is a local cell control flag; see the next point.
3. *conditional operations: control flags in each cell*
   Many of the APSA instructions require masking to determine which cells should perform an operation (this is very similar to masking Processing Elements in the MPP). Each storage cell contains several control flags for this purpose. In addition, some instructions cause each cell to do one of two or three operations; the control flags in a cell determine which operation that cell will perform.

4. *insertion/deletion: conditional shifting hardware*
   A program using APSA can insert or delete a word of information in a data structure in a (small) constant number of cycles. To do this, it executes an instruction that causes a long sequence of words in the storage to shift by one position to the left or right. Such shifts either destroy a word of information (deletion) or they create space for a new word (insertion). The APSA can perform all shifts in one cycle because the storage cells are organized into a hardware shift register. This requires data paths connecting neighboring cells as well as combinational logic in the cells to control the shifting. Not all the cells in the storage take part in these shifts; the local cell control flag values determing what each cell will do during a shift.

5. *pointers: definitions and associative searching*
   The APSA system supports language interpretation by "graph reduction" algorithms, which are generally more powerful and efficient than "string reduction" algorithms because they support shared structures and data recursion. However, graph reduction algorithms require pointer manipulation. A pointer is usually represented by the address of the object that it points to, but that representation would not work in APSA because the shifting operations causes objects to move around in the storage. Consequently each object that is the target of a pointer has a unique "defining identification", and APSA follows a pointer by associatively searching for the corresponding defining identification. The associative searching requires a tree of combinational logic connecting all the storage cells.

6. *searching for values: associative searching*
   The APSA system also makes associative searching available to the user program; this is useful for many applications.

7. *marking sequences of cells: tree logic*
   In order to control the conditional shifting operations, the APSA system needs to set the cells' local control flags correctly. A naive implementation of these flag setting operations would have time complexity proportional to the number of cells in the storage; that is unacceptable. (The local flag setting instructions are actually the most complex operations that APSA performs.) However, a tree of combinational logic whose leaves correspond to the cells is able to perform these operations. This tree also performs other services described above. The system must *simultaneously* compute the local flag values for each cell in the storage. The basic idea behind the logic tree is that many of the local flag computations require some local and some global analysis of the contents of the cells. In general, the lowest node in the tree that has access to all the relevant information for a particular cell computes the local flags for that cell, and many such computations may be made concurrently in distinct nodes of the tree. The nodes that compute local flag values may be at any height in the tree.

These requirements show that the system needs data paths connecting neighboring cells and a binary tree of combinational logic. Figure 1 illustrates the resulting architecture with eight cells. The storage cells are shown as tall boxes containing the corresponding cell numbers, and the combinational logic nodes are squares. Each square contains the interval of accessible cells. For example, the combinational logic node labeled "$0 - 3$" can compute local flag values for cells 0, 1, 2 or 3.

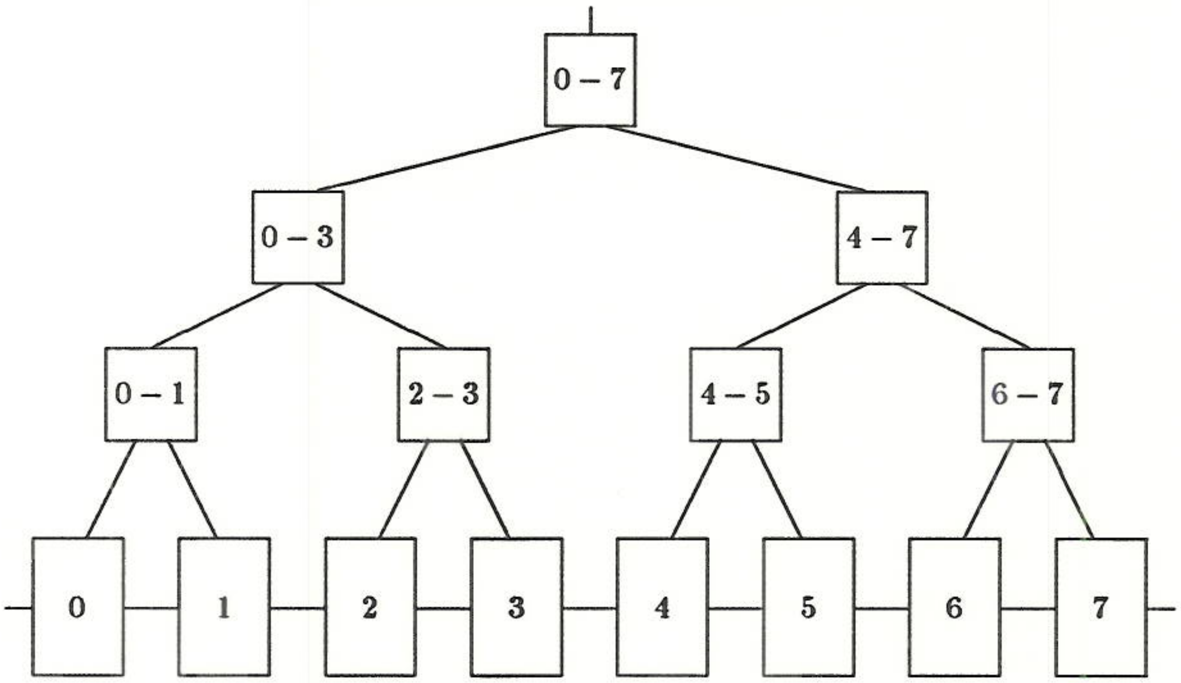


Figure 1. Structure of the APSA system

The APSA will be the main storage for an applicative language implementation, so a realistic system would require at least 10^3 cells. The APSA has more hardware per cell than conventional storage units, but is represents data structures very compactly, so it has greater capacity than a conventional storage with the same number of cells.

It is infeasible to build APSA from conventional small scale integrated chips, because $2 \cdot n - 1$ chips would be needed for n cells of storage. However, Section 8 shows how to lay out the APSA architecture on a VLSI chip, using the available area efficiently. I will use that same technique to "lay out" the APSA tree onto the MPP array unit.

5. Discussion of APSA's Performance

In analyzing APSA's performance while executing an algorithm, we must consider both the number of operations that it must perform and how long each operation takes. The execution time for the algorithm is the product of these two quantities.

APSA provides good performance because it reduces the first of those factors: it executes fewer operations than conventional systems do. In traversing a typical data structure of length n, a conventional applicative language system must follow n pointers by executing $O(n)$ instructions, while APSA only needs to execute a constant number of instructions, which is $O(1)$. Typically the APSA system improves the time complexity of interpreter data structure algorithms by a factor of n.

Analysis of the second factor affecting performance — the time that each operation requires — is more subtle. There are several ways to measure the instruction execution time. *It is important that we use the same measurement criteria for APSA and conventional systems when we are comparing their performance.*

One possibility is to measure the complexity of the instruction execution time with respect to the size n of the data structures, assuming that the total storage size is constant. This obviously leads to the conclusion that instruction execution time is constant — $O(1)$ — for both APSA and conventional systems. Since APSA executes a factor of n fewer instructions than the conventional system, its performance is better by that factor of n.

Another possibility is to measure the complexity of the instruction execution time with respect to the total size N of the storage. Using this measure, each APSA instruction requires $O(log_2 N)$ time, if we consider the gate delays through the combinational logic tree. Similarly, conventional storage units (*e.g.*, RAM chips) require $O(log_2 N)$ time because of the gate delays through their address decoders. (Most RAM chips split their $log_2 N$ bit wide address input into two fields of $(log_2 N)/2$ bits each, and the corresponding address decoders still require $O(log_2 N)$ gate delays.) Therefore, using these criteria, the instruction execution time complexity is again the same for APSA and conventional systems. Thus APSA is again faster by a factor of n (where n is the size of the data structure being processed).

When we consider physical hardware limitations (such as the length of data paths and the limitation of signal speed to the speed of light), both APSA and conventional systems are slower than the analyses given above indicate. Once again, however, the instruction execution time complexity is the same for APSA and conventional systems.

To summarize, the APSA system is faster than conventional systems, often by a factor which is the size of the data structures being processed. APSA gains its speedup by using the overhead time required for address decoding on conventional systems in order to perform the complex logic needed to implement a powerful instruction set.

6. Why simulation of the APSA system is necessary

There are two reasons for simulating the APSA system: (1) to assess the impact of its instruction set on the performance of language interpreters under realistic conditions, and (2) to verify the correctness and performance of the hardware design in order to prepare the way for a possible future VLSI layout.

In order to assess the APSA's instruction set it will be necessary to simulate a language interpreter running on it; it is not enough just to consider how operations on individual data structures would perform in isolation. For example, one advantage of the APSA should be its fast access to variable bindings (in a constant number of cycles). In most language interpreters, actual variable access time is very fast for some variables and slow for others, so a realistic assessment of APSA's variable accessing mechanism requires simulation of the execution of a real program. In addition, storage management (reference counting and garbage collection) should be much faster on APSA than on conventional systems, but a quantitative comparison requires simulation. I plan to simulate several versions of the APSA instruction set, which will help to measure the tradeoffs between the complexity of instructions and their effectiveness. These simulations will be necessary to demonstrate that a VLSI hardware implementation of APSA would be justified.

It is possible to simulate the instruction set at a high level by maintaining an array of records corresponding to the cells. For each simulated cycle, the simulator then iterates through the array, performing all the cell updates and returning the response to the instruction. This method has already proved useful for small experiments [16], but larger experiments require (surprisingly) a lower level simulation. There are two reasons for this:

1. It might appear that the high-level simulation would be faster and therefore preferable. However, the execution time of both the low-level and the high-level simulation will be proportional to the number of storage cells in APSA, and either level of simulation is extremely time-consuming on a conventional computer. The MPP architecture is actually better suited to the low level simulation (see Section 10). Thus the additional time required to do the low-level simulation is not a major drawback.
2. The low-level simulation will give more useful information about the behavior of APSA, and it is essential to have this information before contemplating a

hardware realization. In particular, the simulation will expose any problems in the logic equations, and it may lead to improvements in them.

I have already simulated a small APSA (with 16 cells) at the gate level on a Vax computer, but much larger simulations are infeasible without a massively parallel computer.

7. Objectives of the MPP simulation research

The objectives of the MPP simulation research are to:

1. Provide a testbed for experimentation with variations of the APSA.
2. Measure performance of the APSA in a realistic environment.
3. Develop tools for VLSI layout simulation on the MPP architecture.
4. Assess the capabilities and limitations of the MPP architecture for VLSI simulation.

8. VLSI Layout of the APSA System

This section describes how the tree structure of the APSA system can be mapped onto the square or rectangular area of a VLSI chip. The technique is used on the standard "H Tree" layout. The new feature in the APSA layout is the way the data paths connecting neighboring cells follow the tree structure of the combinational logic node data paths.

The APSA tree architecture is a regular hierarchical layout of subtrees. The layout comprises three kinds of "Computational Element" (CE), which may occur with any orientation. Each CE has a port along one side through which it communicates with the rest of the system. Figure 2 shows each type of CE.

To illustrate the layout method, we shall look at a sequence of APSA trees with increasing sizes. The APSA with tree depth 0 (Figure 3) consists of a single storage cell which contains the "cell storage" field (*cst*) and the "cell combinational logic" field (*ccl*).

A tree of depth k, $k > 0$, consists of two subtrees of depth $k - 1$ connected by a combinational logic node (*ncl*). The physical layout consists of three pieces: the two subtrees are separated by a "corridor" that contains the *ncl* and a data path to the perimeter of the circuit, providing the I/O port. Figure 4 shows the tree of depth 1, where the corridor is simply an *ncl* box.

Larger trees contain longer corridors which must include straight data path nodes (which may, if necessary, contain super buffers). For example, Figure 5 shows the central corridor for $APSA_2$ and the two subtrees.

The individual CEs will correspond to processors in the MPP Array Unit. Figure 6 shows a 3 by 3 array of processors (of course, in the actual simulation the entire array unit will be used), and it shows how the CEs of an APSA with 4 cells are allocated to the MPP processors.

Figures 7, 8 and 9 show increasingly large APSA system layouts. These figures illustrate several interesting properties of the layout method. In particular, note that four orientations of nodes are necessary, but a particular APSA layout needs only two orientations of the cells. Also note that the data paths become longer near the "top" of the tree. The simulation will allocate MPP processors for each of the straight line data paths, so it will be accurate enough to model the delays associated with long distance communications.

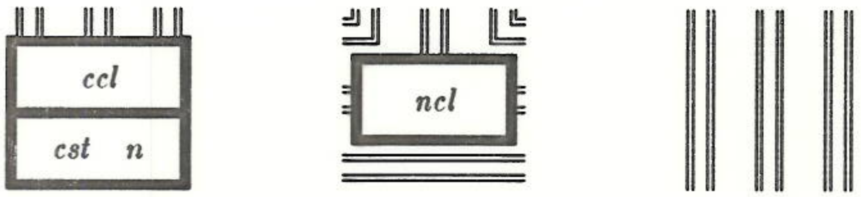


Figure 2. Building Blocks: the Computational Elements

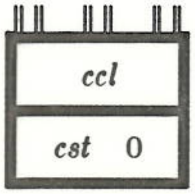


Figure 3. Tree Architecture Layout *(2^0 storage nodes)*

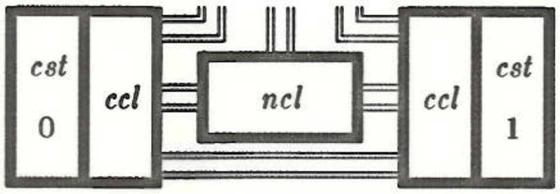


Figure 4. Tree Architecture Layout *(2^1 storage nodes)*

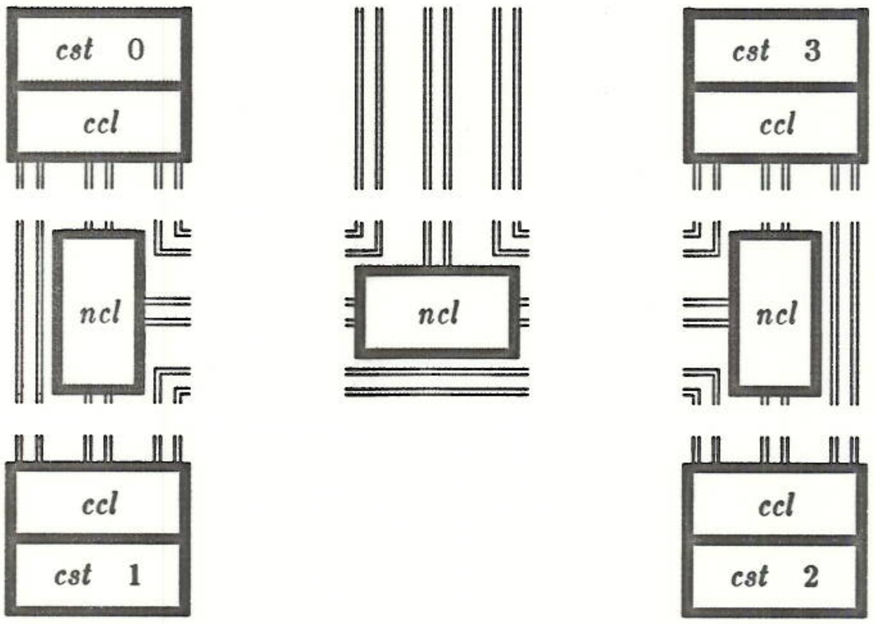
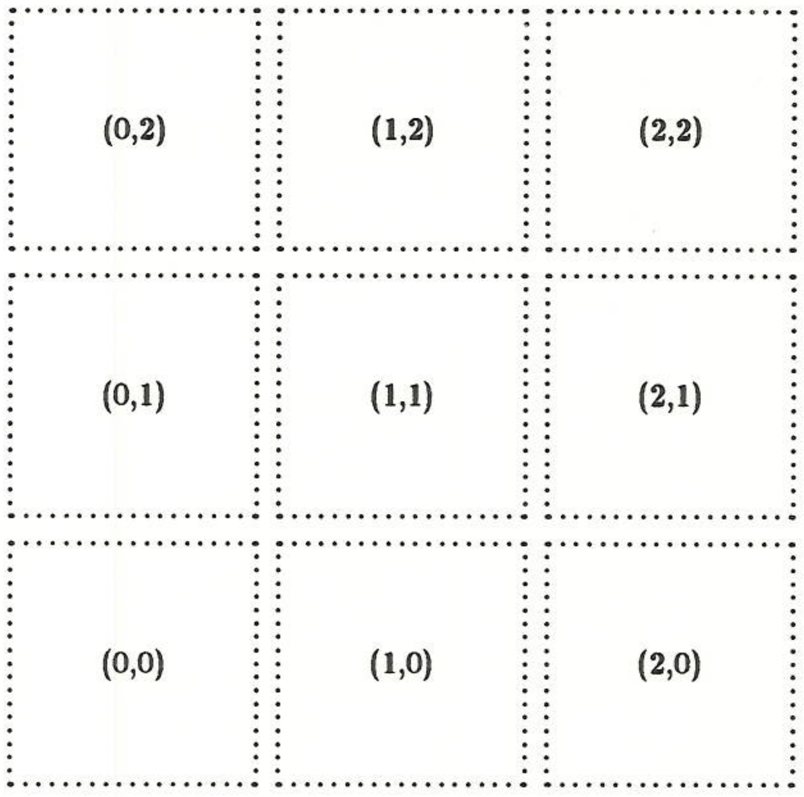


Figure 5. Components of $APSA_2$



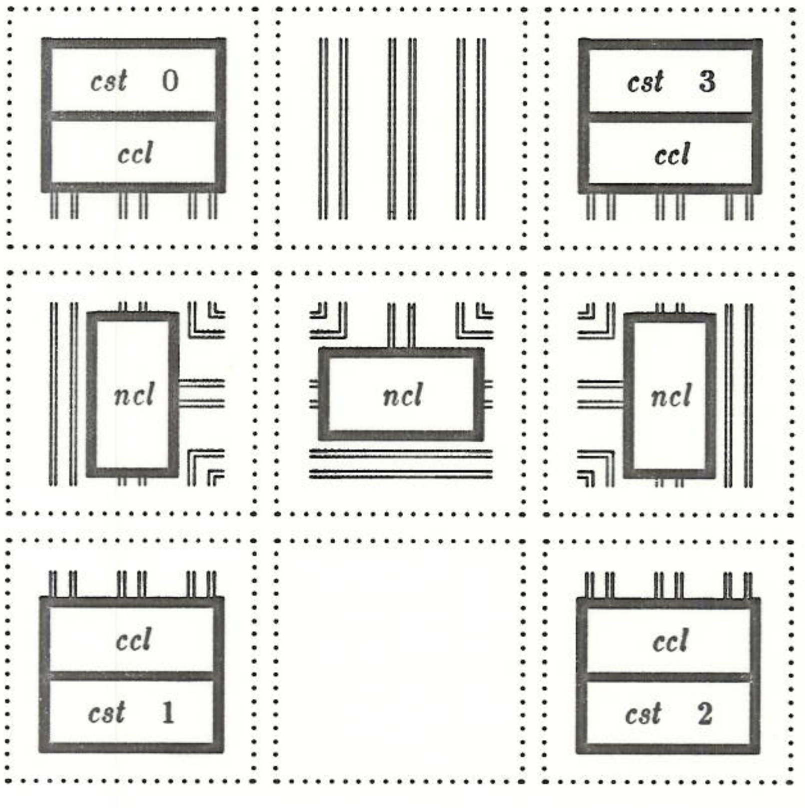


Figure 6. Allocating CEs to MPP processors

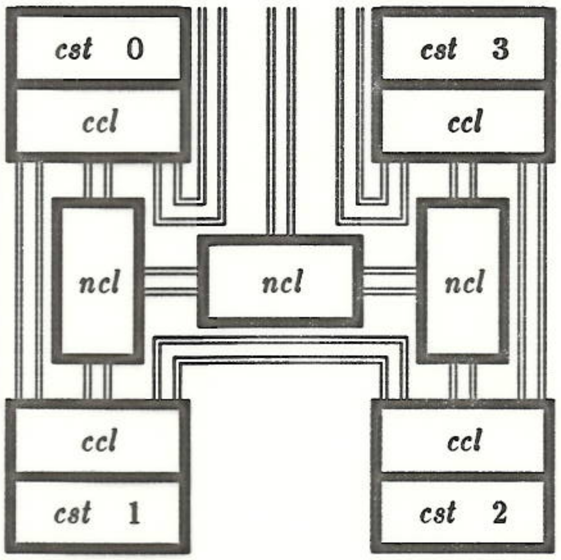


Figure 7. Tree Architecture Layout *(2^2 storage nodes)*

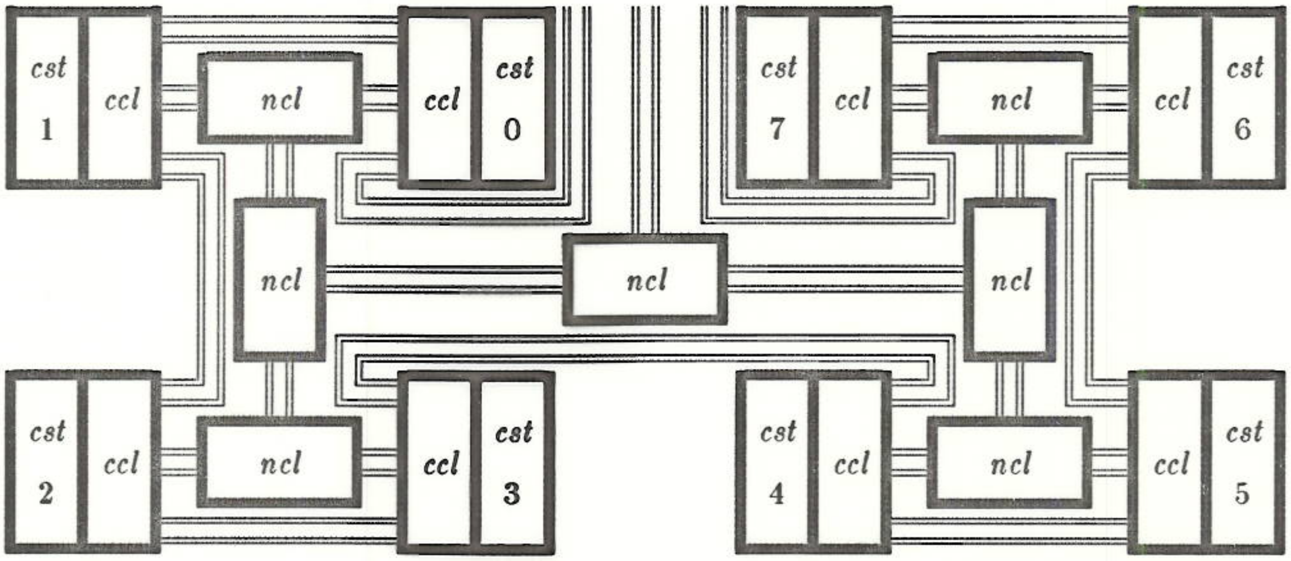


Figure 8. Tree Architecture Layout *(2^3 storage nodes)*

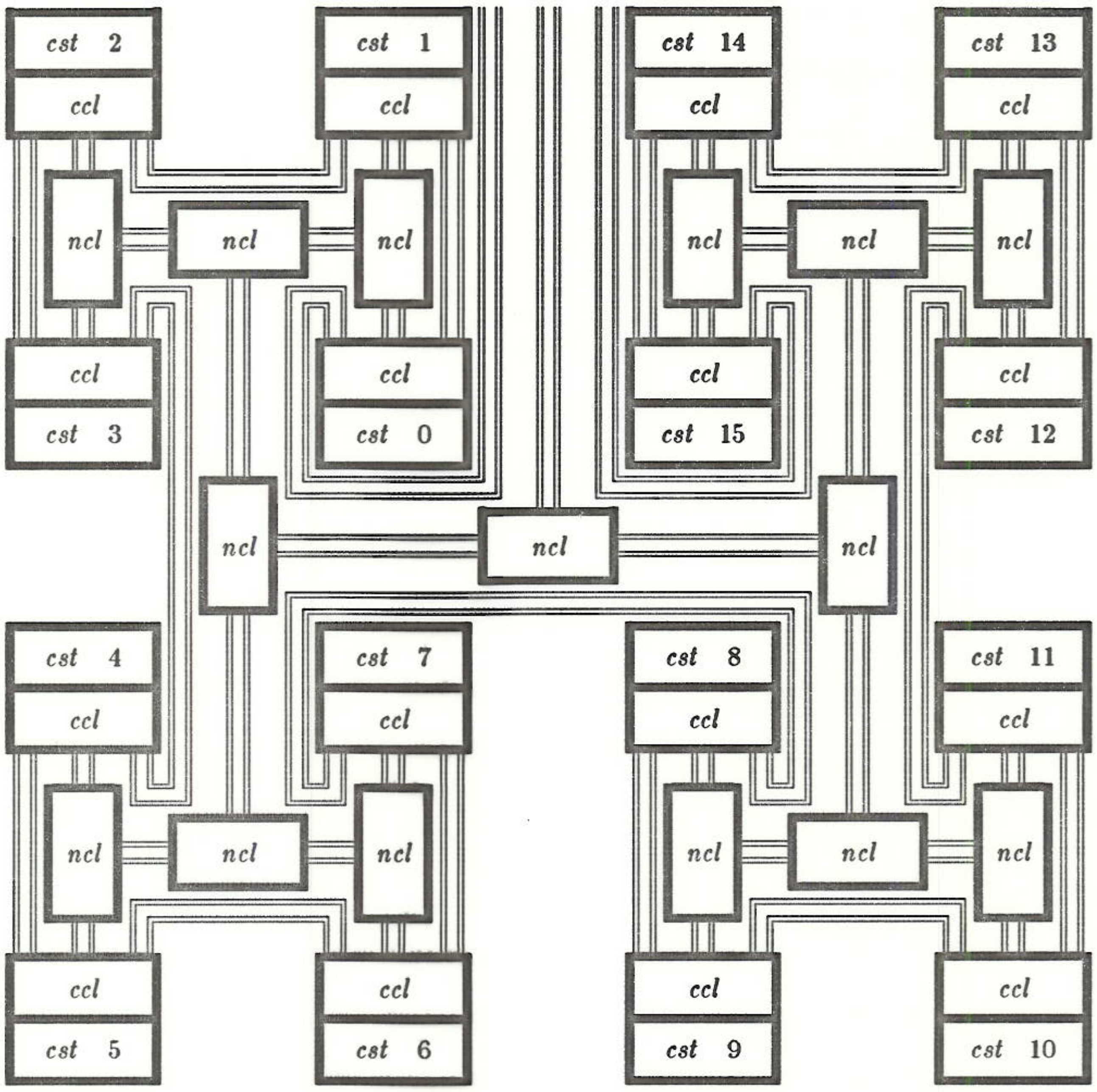


Figure 9. Tree Architecture Layout *(2^4 storage nodes)*

9. Organization of the VLSI Simulator

This section describes a general scheme for simulating VLSI architectures (including systolic architectures) on the MPP. These techniques should be useful for many different kinds of VLSI architecture, although I shall concentrate on simulating just the APSA. Section 8 shows how to generate a VLSI architecture for the APSA tree.

The basic idea of the simulation is that each processor in the MPP Array Unit will correspond to an individual Computational Element (CE) of the APSA layout. The local RAM storage of the Array Unit processors will contain buffers for communication, local storage required by the corresponding CE (this applies only to the APSA storage cell CEs), and a code indicating which type of CE is represented by that MPP processor.

The simulation will consists of a sequence of "simulated APSA clock cycles", which correspond to the APSA instructions. Each simulated APSA clock cycle is a sequence of "VLSI box transmission periods", which represent the time required for a CE to read its inputs and produce its outputs. Each VLSI box transmission period, in turn, will require a sequence of MPP Array Unit instructions that perform the operations required by the CE, bit by bit.

Each simulated VLSI box transmission period will begin with MPP instructions that cause each processor to read the outputs of its four neighbors and store them in the local RAM storage. Next, the simulator will issue instructions that cause the processors to compute the new local storage values (if any) of the associated CE, and the output values. By repeating these operations, the MPP Array Unit will simulate the flow of information through the APSA system.

I plan to microprogram the simulator, because Parallel Pascal (the only high level language currently available for the MPP) does not provide all the operations needed for an efficient simulation. The APSA architecture will be specified by a set of tables, and the simulator will be driven by those tables. Consequently the simulation tools will be applicable to other VLSI simulations. This project will probably result in MPP software that is applicable to many VLSI simulation problems.

10. How the APSA simulator will exploit the MPP's capabilities

I plan to simulate APSA at the gate level (Section 6 discusses the reasons for performing such a low level simulation). The architecture of the MPP has exactly the right combination of processors, storages and interconnections for the simulation.

The simulation techniques will also be applicable to other VLSI designs, and will be extremely useful for studying systolic architectures [11] in general. VLSI architecture simulation is notorious for its computation requirements. Therefore the results of this work will be widely applicable in the very active area of VLSI architecture.

Hardware designs are sometimes specified simply as an interconnection of components. What distinguishes a VLSI architecture is its geometrical layout: the *positions* of all components and interconnections must be specified, and the entire circuit should use a rectangular chip area efficiently.

Since the APSA has a tree structure, the traditional "H Tree" layout [15] efficiently maps it onto a square or rectangle. It is interesting (and crucially important) that the data paths that connect neighboring cells also fit into the H Tree layout, although they do not correspond to tree edges themselves. Section 8 describes how the APSA H Tree layout works, and it shows both the tree edge data paths and the neighboring cell data paths.

The layout consists of a grid consisting of "CEs", or computational elements: combinational logic nodes (called *ncl*), leaf cells containing both storage and combinational logic (called *cst*, for cell storage), straight-line data paths, and empty spaces. In addition, the CEs (except for empty spaces) all come in several orientations. The orientation of a CE is significant in VLSI designs because it can affect the connections of the transistors with the power supply and ground.

The simulation will map a square array of CEs onto the square array of processors in the MPP. Each MPP processor will simulate an individual CE. In order to coordinate the activities of all the CEs, each MPP processor local memory will contain several bits that describe which kind of CE it represents — that information suffices to specify what its inputs are, how it will process them, and what outputs it will produce. The MPP will broadcast instructions that cause all the MPP processors corresponding to a specified type of CE to perform their CE functions simultaneously. Several such broadcasts for all the types of CE will cause the entire array to simulate a small time interval. A sequence of these time interval simulations will allow values to propagate through the long straight line data paths, the nodes, and the cells; such a sequence will simulate an APSA clock cycle.

It is interesting to note how well the MPP architecture fits onto this simulation scheme (which will also work for other types of VLSI simulation). First, each APSA CE has connections only to its four neighbors (north, south, east and west). Those connections correspond to the nearest neighbor connections in the MPP. (Many VLSI designs use only these interconnections; other VLSI architectures, such as hex-connected systems, would require additional communication overhead.) Second, this simulation exploits the fact that the MPP is a bit serial machine. The simulation would not make use of floating point functional units, if the MPP had them, but it does need to have local storage for each CE, a boolean function logic box, a bit adder, a shift register, and connections to the four nearest neighbors. The MPP processors provide exactly these requirements. As a result, the VLSI simulation will run much faster on MPP than it would on most supercomputers, including vector processors.

11. Why the MPP is necessary for this research

In order to meet the objectives outlined in Section 7, I will need to perform a number of simulations of the APSA system running small applicative program examples. This will be necessary both to analyze the behavior of the data structures under various conditions, and to experiment with variants of the APSA instruction set.

An order-of-magnitude analysis of the computation requirements of a simulation shows that conventional computers are totally inadequate. A realistic simulation will require 10^3 storage cells (a smaller number could not handle non-trivial applicative programs). An APSA with 10^3 cells contains 10^4 computational boxes, each of which must be simulated individually. (In particular, for an APSA with 4096 cells there will be a VLSI array comprising 127 by 127 computational boxes, which almost fills the MPP processor array.) Each computational box will need 10^2 or 10^3 individual bit operations per simulated cycle (depending on the particular version of the APSA chosen). Finally, simulation of a program running on APSA will require up to 10^5 cycles. Thus each simulation may require $10^{4+3+5} = 10^{12}$ bit operations. The MPP executes 10^7 cycles per second, and performs 10^4 bit operations per cycle, so it can do 10^{11} bit operations per second. This leads to an expected simulation time on the order of 10 seconds. In contrast, the expected simulation time on a Vax computer is 10^6 seconds, or about 280 hours.

Experimental analysis and development of the APSA architecture is impossible on a conventional machine. Since the MPP has exactly the right interconnection among processors for VLSI simulation, and since its bit-serial operations correspond to the individual operations within an APSA computation box, the simulation will fully exploit its massive parallelism. Only the MPP or a similarly massive parallel processor will be able to support experimental simulation research on VLSI architectures like APSA.

12. Significance of the expected results

The simulation research outlined above is expected to produce the following results:

1. It will demonstrate that the MPP architecture can simulate VLSI architectures extremely efficiently.
2. It will result in programming techniques and software that will be be useful for other VLSI simulations.
3. It will lead to improvements in the APSA architecture.
4. It will provide independent confirmation of the correctness of the APSA logic equations (or help to debug them).
5. The experimentation that it supports may lead to further advances in architectures for applicative programming.

It is significant that the research described in this paper is extremely well-suited to the MPP architecture, yet it is quite different from the numerical image processing applications usually associated with the MPP. This work may enlarge the domains of research deemed suitable for MPP, which is especially important for a special-purpose supercomputer.

References

1. John Backus, "Can Programming be Liberated from the von Neumann Style?", *CACM* Vol. 21, No. 8, 1978.

2. C. G. Bell and A. Newell, *Computer Structures: Readings and Structures*, McGraw-Hill, New York, 1971.

3. B. L. Buzbee and D. H. Sharp, "Perspectives on Supercomputing", *Science*, Vol. 227, No. 4687, 8 February 1985, pp. 591–597.

4. Will Clinger, "A Scheme311 Compiler: An Exercise in Denotational Semantics", *Proceedings of the 1984 LISP Conference*, ACM, New York, 1984.

5. D. Friedman, C. Haynes, E. Kohlbecker and M. Wand, *Scheme 84 Interim Reference Manual*, Technical Report 153, Computer Science Department, Indiana University, Bloomington, In., January 1985.

6. Friedman and Wise, CONS Should Not Evaluate its Arguments, Automata, Languages and Programming, Edinburgh Univ Press, 1976.

7. Paul Hudak and Adrienne Bloss, "The Aggregate Update Problem in Functional Programming Systems", *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985.

8. Steven D. Johnson, Circuits and Systems: Implementing Communications with Streams, Proc. of the 10'th IMACS Symposium on Systems Simulation and Scientific Computation, Vol.5, 1982.

9. Donald E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Second Edition, Addison-Wesley Pub. Co., Reading, Mass., 1973.

10. Anne Kohlstaedt, Daisy 1.0 Reference Manual, Technical Report 119, Computer Science Department, Indiana University, Bloomington, 1981.

11. H. T. Kung, "Let's Design Algorithms for VLSI Systems", *Proceedings of the Caltech Conference on Very Large Scale Integration*, California Institute of Technology, Pasadena, 1979.

12. Gyula A Mago, "A Network of Microprocessors to Execute Reduction Languages" (Parts 1 and 2), *International Journal of Computer and Information Sciences*, Vol. 8, No. 5 (Part 1) and No. 6 (Part 2), 1979.

13. Gyula Mago, "Data Sharing in an FFP Machine", 1982 LISP Conference, August 1982.

14. John McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine", *Communications of the ACM*, April, 1960.

15. Carver Mead and Lynn Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

16. John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Ph. D. Thesis, TR81-5, University of Iowa, Iowa City, 1981.

17. John T. O'Donnell, "Dialogues: A Basis for Constructing Programming Environments", *1985 SIGPLAN Symposium on Programming Languages and Programming Environments*, 1985.

18. John T. O'Donnell, "An Efficient Architecture for Implementing Sparse Array Variables", *Proc. Twenty-Third Annual Allerton Conference on Communication, Control and Computing*, October, 1985, to appear.

19. John T. O'Donnell, "An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages", *IFIP Symposium on Functional Programming Languages and Computer Architecture*, Nancy France, 1985, to appear.

20. Jerry L. Potter (ed.), *The Massively Parallel Processor*, The MIT Press, Cambridge, Ma., 1985.

21. S. R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984.

22. David S. Wise, "The Applicative Style of Programming", *Abacus*, Vol. 2, No. 2, Springer-Verlag, New York, Winter 1985.

23. David S. Wise, *Design for a Multiprocessing Heap with On-Board Reference Counting*, Technical Report 163, Computer Science Department, Indiana University, Bloomington, 1985.