AN OPTIMALLY PORTABLE SIMD PROGRAMMING LANGUAGE

Russ Tuck
Computer Science Department, Duke University, and
Computer Science Department, University of North Carolina at Chapel Hill *

ABSTRACT

Existing programming languages for SIMD (Single-Instruction Multiple-Data) parallel computers make implicit architectural assumptions. These limit each language to architectures satisfying its assumptions. This paper presents a theoretical foundation for developing much more portable languages for SIMD computers. It also describes work in progress on the design and implementation of such a language.

An *optimally portable* programming language for a set of architectures is one which allows each program to specify the subset of those architectures on which it must be able to run, and which then allows the program to exploit exactly those architectural features available on all of the target architectures. The features available on an architecture are defined to be those the architecture can implement with a constant-bounded number of operations. This definition ensures reasonable execution efficiency, and identifies architectural differences which are relevant to algorithm selection.

An optimally portable programming language for SIMD computers, called Porta-SIMD (porta-simm'd), is being developed to demonstrate these ideas. Based on C++, it currently runs on the Connection Machine and Pixel-Planes 4.

Keywords: Portable, SIMD Parallel, Programming Language, Porta-SIMD, Taxonomy, Pixel-Planes, Connection Machine, C++.

INTRODUCTION

Portable high-level languages for von Neumann computers are major accomplishments in computer science. These languages have radically improved the quality, cost, reliability, and availability of software. However, the greater architectural diversity of SIMD (Single-Instruction Multiple-Data) computers has so far kept them from fully benefiting from such languages. Each existing SIMD language contains architectural assumptions which make it suitable for programming only a certain subset of SIMD machines.

Optimal portability is a new concept which can guide the development of much more portable SIMD programming languages. It is based on the recognition that some differences among SIMD architectures significantly influence algorithm selection. These should not be completely hidden from the programmer.

The programmer makes an algorithm's architectural assumptions explicit by expressing the algorithm as a program for a particular set of architectures. These architectural assumptions precisely define the program's portability. The programmer may then take full advantage of all architectural features common to all members of that set, and no more. Selecting a small set of very similar architectures limits a program's portability, but allows it to take full advantage of specialized features the members share. Selecting a large diverse set of architectures produces a program that is very portable, but may not take full advantage of some of the architectures. This selectable tradeoff between breadth and power provides optimal portability.

*This work was supported by the Pixel-Planes Project, Henry Fuchs and John Poulton, P.I.s, and its grants: National Science Foundation grant #MIP-8601552, Defense Advanced Research Projects Agency order #6090, Office of Naval Research contract #N0014-86-K-0680; and by the GRIP Project, Frederick Brooks, P.I., under National Institutes of Health grant #RR 02170. Access to a Connection Machine was provided by the Advanced Computing Research Facility at Argonne National Laboratories, under grants NSF-ASC-8808327 and DOE-W-31-109-ENG-38. Author's phone and electronic address: (919) 962-1755 or (919) 684-5110; tuck@cs.unc.edu or rrt@cs.duke.edu.

This is entirely consistent with Chandy and Misra's (Ref. 8) ideas on algorithm portability. They advocate developing algorithms that are progressively more tightly bound to particular architectures, until an algorithm is specialized sufficiently to provide the desired performance. They provide a language-independent notation for expressing algorithms during development, which must be translated into a language for a particular architecture before execution. With an optimally portable language, this would not have to be a different language for each target architecture. Avoiding the necessity of learning and remembering details of a different language for each architecture is a significant time and cost savings.

In practice, an optimally portable language for a set of architectures needs both a definition and a taxonomy of that set. These provide a precise way to specify the architectures on which a program must run. They also contribute to improved understanding of the architectures, and their algorithms and languages. Both a definition and a taxonomy of SIMD architectures are given in the section "A SIMD Taxonomy for Programmers."

Existing SIMD programming languages are not optimally portable. They are built on a variety of inflexible architectural assumptions, including specific processor interconnection networks and the presence or absence of features like local addressing of memory. The section titled "Existing SIMD Languages" surveys these languages.

I am currently working on the design and implementation of a new optimally portable language for SIMD computers: Porta-SIMD (pronounced porta-simm'd). Its overall structure is modeled on the proposed SIMD taxonomy for programmers, allowing it to present to the programmer an appropriate programming model for any subset of SIMD architectures. It is intended to demonstrate the feasibility of designing, implementing, and using optimally portable languages. The ongoing design and implementation of Porta-SIMD are discussed in the section "An Optimally Portable Language."

OPTIMAL PORTABILITY

Optimal portability is best defined in terms of a few supporting definitions. An *abstract architecture* is the set of fundamental data types and operations provided by a computer, without regard to how the data and operations are represented. It does not include implementation details such as the the amount of memory present in a machine, or the number of processors in a parallel machine. Except where explicitly stated otherwise, I will use *architecture* as a synonym for abstract architecture.

The members of a set of architectures are *equivalent* if and only if their intersection is identical to their union. The *union* of a set of architectures is an architecture containing all data types and operations contained in any member of the set. The *intersection* of a set S of architectures is an architecture constructed as follows:

1. Let architecture u be the union of S. To each member A_i of S add each data type and operation in u which A_i can simulate with a constant number of its own data elements and operations.
2. Take the intersection of the sets of data types and operations of all members of S, as augmented by the previous step, to create the intersection architecture.

The intersection of a set of architectures will also be called the *shared architecture* of the set. These definitions imply that any member of a set of equivalent architectures can simulate the operation of any other member, and the number of native operations they execute will be within a constant factor of each other.

A particular computer may be considered to implement only a single set of equivalent architectures. This set must be the set of architectures equivalent to the architecture defined by the computer's lowest-level publically documented programming interface. For most sequential computers, that interface is assembly language. For some SIMD computers it is a library.

A program is *portable* across a set S of architectures if and only if it can be compiled and correctly executed on the shared architecture of S. Such a program can therefore be compiled and correctly executed on every member of S. The architecture on which a program is intended to run is called the program's *target architecture.* A program is said to *use* a data type or operation if and only if it contains a direct or indirect reference to a language feature that provides a capability equivalent to that data type or operation.

A programming language L is *optimally portable* for a set S of architectures if and only if all of the following are true:

- L requires each program p to specify some architecture $A_p \in S$ as its target architecture. (A default target architecture may be implicitly specified in the absence of an explicit specification.)
- L does not allow p to use any data type or operation not in A_p.
- L allows p to use any data type or operation in A_p.

This definition implies that p is portable across any set $S_1 \subseteq S$ such that A_p is the shared architecture of S_1, including the maximal such set, S_p. Therefore, p cannot be portable across a larger set of architectures without giving up the use of one or more data types or operations. In addition, p cannot use additional data types or operations without adding to A_p. This would potentially reduce p's portability by removing architectures from S_p.

A few points in the definition of optimal portability deserve discussion. It is difficult, perhaps impossible, to find a simple set of rules to accurately and impartially determine the programmer-visible architecture of every computer. Computer systems have many layers of architecture, and features are sometimes implemented in the "wrong" layer conceptually to improve performance. However, identifying such features is a matter of judgement which is not easily reduced to simple rules. Great care has been taken in constructing the definitions above, but they are not perfect.

It is important to construct a good test for whether an abstract architecture can usefully simulate some data type or operation. Any Turing-equivalent machine may simulate any architecture, but not always with useful performance. The constant-bounded criterion above for operations and data ensures reasonable performance and fits well with intuitive notions of equivalent architectures. It also makes equivalence transitive. (Suppose architecture A_x can simulate architecture A_y in $op(A_x, A_y)$ operations, and equivalence is denoted by "=". Then $A_i = A_j$ and $A_j = A_k$ implies $op(A_i, A_j) \leq op(A_i, A_j)op(A_j, A_k)$, which implies $A_i = A_k$ because $op(A_i, A_j)$ and $op(A_j, A_k)$ are constants.) Logarithmic and polynomial bounds do not have this important property.

In some cases, a single machine may be reasonably described by two or more quite different abstract architectures. As long as they are equivalent, they are equally valid descriptions. For example, a bit-serial SIMD machine may be described as having operations on bits, on multi-bit integers, or on floating-point numbers. Operations on the multi-bit data types can be simulated by a constant number of bit-serial operations. The constant (which may be over 1000) depends on the nature and size (in bits) of the simulated data type, but does not depend on the values stored in data elements of that type. The architectures are equivalent. This is consistent with the common practice of building implementations of a single architecture with varying execution speeds.

Another example is a SIMD machine with a 2-dimensional grid interconnection network which allows communication in parallel between pairs of adjacent PEs (Processing Elements), using its lowest-level publically documented programing interface.. With an additional layer of software to do automatic routing, it might also be described as providing communication between arbitrary pairs of PEs. The number of operations required to simulate arbitrary communication with this network depends heavily on the dynamically chosen communication pattern. A lower bound for the worst case is the diameter of the network, which is at least the square root of the number of PEs. Since a SIMD architecture does not specify a maximum number of PEs, this is not a constant bound. Therefore, the two descriptions are not equivalent, and only the first is part of a valid abstract architecture for this machine.

However, if the automatic routing software were hidden beneath the lowest-level publically documented programming interface, the architecture would be considered by the above definitions to provide communication between arbitrary pairs of PEs.

There are several reasons to define a machine's architecture by its lowest-level publically documented programming interface, rather than by its hardware. A programmer has no access to the hardware except through this interface. Hardware documentation is not always publicly available; it is often less complete and precise than the programming interface, largely because programming interfaces must be well documented in order for important software to be developed. Machine builders are free to implement a single architecture with different hardware designs, transparently to the programmer. These identically programmed machines should be considered to have the same architecture (from a programmer's perspective).

It is difficult to define precisely which data types and operations a program uses. The important feature of the definition of *use* above is that usage is defined with respect to the source code, not the compiled object code. This prevents the compiler from making features not available in the target architecture available to the program by generating code to simulate them with arbitrary numbers of data elements and operations. (Of course, a compiler generating code for an architecture equivalent to A_p may generate a constant number of data elements and operations to simulate data types and operations of A_p.)

Prohibiting compilers from simulating data types and operations not present in A_p ensures portability with useful performance, not just theoretical portability. This does not restrict the function of programs, since p may simulate such data types and operations itself. The implementers of L may even provide, as a convenience to programmers, a package written in L to do this simulation.

A SIMD TAXONOMY FOR PROGRAMMERS

A programming language is optimally portable only for a specific set of architectures. Therefore, any optimally portable SIMD programming language will require a definition of SIMD architectures.

Definition of SIMD Architectures

An architecture A is a SIMD architecture if and only if all of the following are true:

- A has a host computer which handles ordinary scalar computations and flow control, and which broadcasts instructions, one at a time, to all PEs (Processing Elements).
- A has $n > 1$ identical PEs which all execute, simultaneously, each instruction broadcast by the host.
- Each PE is able to evaluate basic arithmetic and logical expressions.

I believe every useful SIMD architecture also has the following properties:

1. Each PE is able, in response to broadcast instructions, to independently choose whether to ignore instructions to modify its memory. (PEs executing all instructions are *enabled,* while those ignoring instructions to modify memory are *disabled.* PEs can be considered to have an *enable-bit* which is 1 only in enabled PEs.)
2. Each PE is able to compute its unique PE number $0 \leq p < n - 1$, given sufficient time.
3. Each PE has its own private memory.

Property 1 can be simulated with a constant number of ordinary arithmetic and logical operations. Architectures that do not have this property are therefore equivalent to those that do, and can be considered to have it. This property takes many different but equivalent forms in various machines, with it being possible to ignore different subsets of an instruction set.

Property 2 certainly holds for all architectures which have a connected communication graph, and which allow any single PE to be distinguished in any way. It also holds for all architectures with parallel input, since the data being read can be the PE numbers. Property 2 holds if an architecture can load into each PE a different element of a set of distinct values, by any means, since this set can be the PE numbers. If there is a SIMD architecture which does not have this property, I do not think it is very interesting because the PEs cannot be given unique predetermined data on which to operate. That is the whole purpose of a SIMD architecture.

The only claimed exception to property 3, that I am aware of, is an alternative set of architectures where PEs access a global memory space through a network of some kind (e.g., (Ref. 20, pp. 326-327)). I believe that any such architecture is equivalent to a local-memory architecture in which the PEs are connected to each other by the same network that connects the PEs to the global memory.

Specifically, the BSP (Burroughs Scientific Processor) (Ref. 20, pp. 326-327, 410-422) is the only non-local memory architecture I know of. It is equivalent to a large subset of the CM (Connection Machine) architecture (Refs. 18, 10, 1). (Both architectures are discussed briefly in a later section.) The BSP can simulate the CM simply by assigning a distinct portion of global memory to each PE for private use, and accessing memory assigned to other PEs only to simulate communication. Similarly, the CM can simulate the BSP by using its communication primitives to access memory, treating all the private memory as a single global memory space. Both simulations take constant time, so the BSP's global memory and arbitrary PE to memory interconnection network is equivalent to the CM's local memory and a subset of its communication primitives. The only difference between the architectures is that the CM has somewhat more powerful mechanisms for resolving simultaneous accesses to a single memory location.

If any of these properties is not true of all SIMD architectures, then the taxonomy below is considered to have an additional dimension for each such property. Because all architectures currently classified by this taxonomy have the same coordinates along these dimensions, those coordinates will not be mentioned further.

Taxonomy of SIMD Architectures

An optimally portable SIMD programming language must recognize and handle the full diversity of SIMD architectures that exist within this definition. A taxonomy of SIMD architectures will be crucial to this task. Although many architectural differences can be almost completely hidden by a high-level language, others fundamentally influence the programmer's algorithm selection. To be most useful for portable language design, the taxonomy should exclude the former and focus on the latter. The differences that do not influence algorithm selection can be uniformly hidden from the programmer by language abstraction. However, an optimally portable language must make the remaining differences visible to the programmer, in the form of language features which exploit the target architecture.

Previous SIMD taxonomies have been constructed with different goals, and consider some architectural features which need not be visible to a programmer. Examples include work by Hwang and Briggs (Ref. 20, chapters 5-6), and a tutorial by Seitz (Ref. 32). Fountain (Ref. 13) and Gerritsen (Ref. 16) compare certain SIMD implementations at a level appropriate for system designers and architects, rather than programmers. An extended abstract by Jamieson (Ref. 21) considers matching algorithms with all kinds of parallel architectures, not just SIMD. Karp (Ref. 22) presents a taxonomy restricted to "those aspects that affect coding style," but considers only MIMD (Multiple-Instruction Multiple-Data) architectures. These taxonomies not suited for designing an optimally portable SIMD language.

Beginning with the most important, the architectural differences that can significantly influence algorithm selection include:

Topology — the labeling and adjacencies of the PEs;

Communication — whether each PE can read/write data to/from (0) no other PE, (1) a globally-selected adjacent PE, (2) a globally-selected location in a locally-selected adjacent PE, or (3) a locally-selected location in a locally-selected adjacent PE;

Collision Resolution — whether multiple writes to the same location under communication types (2) and (3) are resolved by (0) serializing the accesses, or (1) combining them by applying an arithmetic or logical operation;

Local Addressing — whether local PEs' memories can be addressed (0) only by a single globally computed address, or (1) also by addresses computed locally at each PE;

Global Logical-Or/Multiple-Response Resolver — whether the host can determine in a constant number of operations (0) neither of the following, (1) if any PE has a non-zero value in a certain field of memory (global logical-or), or (2) the identity of at least one PE having a non-zero value in a certain field of memory, if such a PE exists (multiple-response resolver);

Parallel I/O (Input/Output) — whether it is (0) impossible or (1) possible for all PEs to transfer data to and from a mass storage subsystem in parallel;

PE to Host I/O — whether the host can obtain data from (0) no PE, (1) only a subset of PEs, or (2) any selected PE.

These architectural differences define a discrete 7-dimensional space. A SIMD architecture can be characterized by a 7-tuple giving its location in this space. All the dimensions except the first, topology, have a finite set of values enumerated in their descriptions above. As new SIMD architectures are developed, it may be necessary to add new dimensions to this taxonomy to accomodate newly invented architectural features.

Topology and communication are very closely related. Without inter-PE communication, all topologies are equivalent. However, a SIMD architecture without inter-PE communication may still use a particular topology. The 2D topology of Pixel-Planes (discussed below) is a good example. The (x, y) labeling and adjacency of PEs are necessary to evaluate bilinear expressions, and to map computed values from PEs to pixels.

In both communication and local addressing, local selection subsumes global selection, since it is trivial to make the same local selection at all PEs.

Communication type (3) provides local addressing as a side effect. It would be conceptually cleaner to eliminate this communication option and allow it to be simulated by communication type (2) and local addressing. This was not done because the simulation takes operations proportional to the maximum number of access to any one PE, and because communication type (3) is a single operation of the CM and BSP. However, both these machines essentially perform the same simulation in hardware or microcode. This is an example of an operation moved down a layer in the architecture for performance reasons. It exposes a limitation of the methods used here to delineate programmer-visible architectures.

Global logical-or has several equivalent variants. These include the similar "global logical-and", and the related special case "all enables off", which is the inverse of global logical-or applied to the bit which determines whether local memory is write-protected.

This taxonomy has not yet been extended to include two architectural features. The first is cut-through routing of data between PEs. Cut-through routing allows some PEs to send data to non-adjacent PEs, provided the intervening PEs do not send data. The Princeton Engine (Ref. 9) and the ASP (Associative String Processor) (Ref. 23), both 1D architectures, use this.

The second feature is performing parallel-prefix as a single operation. The CM provides this capability, though the microcode must simulate it in a number of operations logarithmic in the number of PEs involved. (This can be proven, since each PE can only combine two values in a single operation.) This is another example of an operation moved down a layer in the architecture for performance reasons.

This taxonomy of SIMD architectures specifically excludes a variety of differences which may be very important to computer architects, but which need not influence algorithm selection. Among these are word length, memory structure and size, special hardware for floating-point operations, and details of scalar and parallel machine instructions. These are all routinely hidden by the abstractions of ordinary high-level languages, and handled by compilers. Of course, the hiding is sometimes imperfect, and it is possible to write non-portable programs which depend on word length, byte order, or other machine-specific details. However, a few simple coding rules are generally sufficient to avoid these problems. Neither the problems nor the solutions differ fundamentally between sequential and SIMD-parallel architectures. SIMD languages should be able to hide these architectural differences as well as, but not necessarily better than, sequential languages.

Figure 1 represents as a tree the space of SIMD architectures defined by the proposed taxonomy. The labels on the left identify the dimension of space represented by each level of branching. The label at each interior tree node identifies the location of the subtree rooted at that node along one dimension of architectural space. Leaf nodes represent selected published SIMD architectures. Subtrees containing no selected architectures are not shown. The

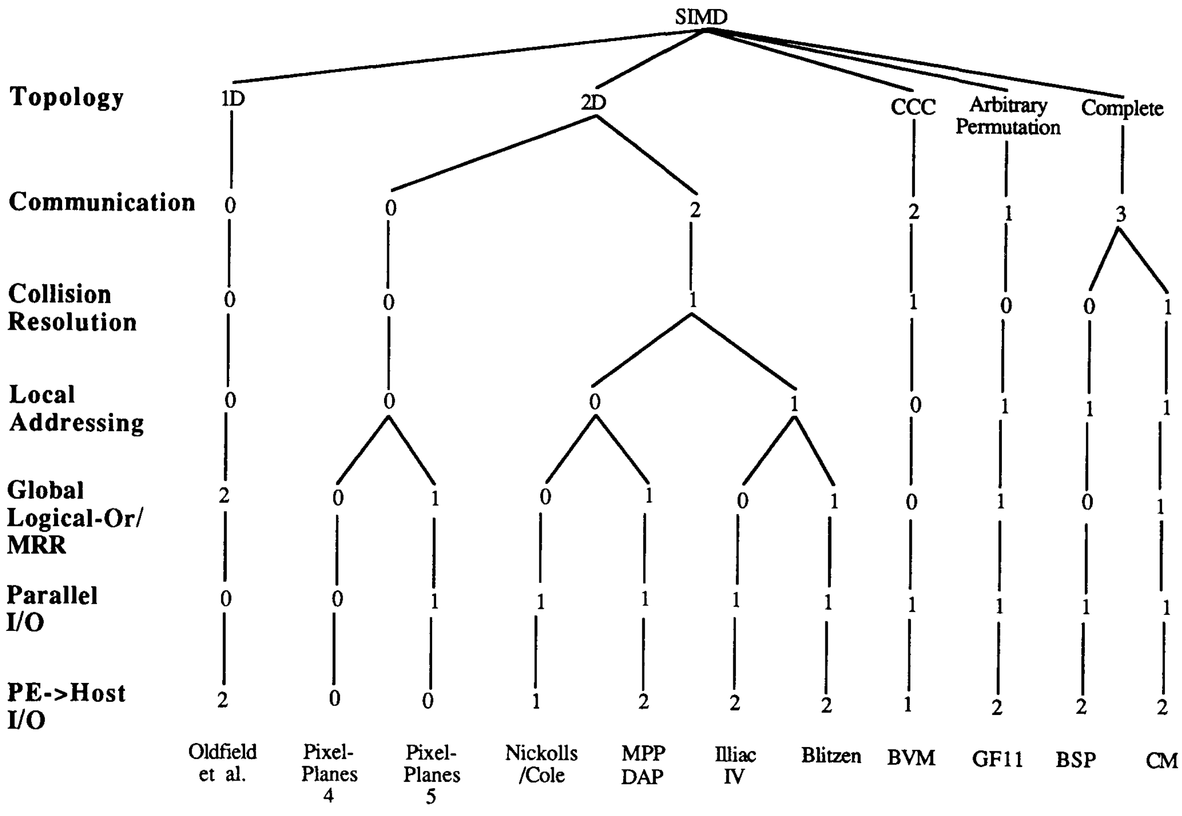

Figure 1. SIMD Architectures

space available is not sufficient for the entire set of SIMD architectures, so I have included as representative a variety as possible. Additional references are always welcome.

This taxonomy has the desirable characteristic that it is easy to determine that certain architectures are subsets of others. This is useful because programs for a particular architecture are portable to all supersets of that architecture. The enumerated dimensions all obey a strict subset ordering. Therefore, one architecture is a subset of another if they have the same topology and if each of the remaining elements of the first 5-tuple is no greater than the corresponding element of the second 5-tuple. For example, the MPP (2D, 2, 1, 0, 1, 1, 2) is a subset of BLITZEN (2D, 2, 1, 1, 1, 1, 2), but not of Pixel-Planes 4 (2D, 0, 0, 0, 0, 0, 0).

In a few special cases, an architecture may fail this criterion and yet be a subset of another. Examples include the following:

- For topologies with a constant number of neighbors per PE, local and global selection of neighbors for communication are equivalent. Collision resolution by serialization or combination are also equivalent for these topologies. Of the topologies discussed below, 1D , 2D, and CCC have a constant number of neighbors per PE, but Hypercube, Arbitrary Permutation, and Complete do not.
- Communication type (3) effectively provides local addressing type (1).
- Global logical-or effectively provides arbitrary PE to host I/O (2).
- An architecture which has parallel I/O to a random access storage device which the host can also manipulate, but does not have PE to host I/O, can simulate arbitrary PE to host I/O. A second architecture differing from the first only in having PE to host I/O and lacking parallel I/O is therefore a subset of the first.

In each case, the result is that adjacent points in architectural space are related by the equivalence rather than the subset relation.

Survey of SIMD Architectures

Most of the remainder of this section surveys the SIMD architectures appearing in figure 1. It shows how they fit within the space of the proposed taxonomy, giving evidence that the taxonomy is reasonably complete. For simplicity, each architecture is described as if it were the equivalent canonical architecture defined by its location in architectural space. The proofs of equivalence are generally not difficult, but will not be presented here. The architectures will be treated in order from left to right across the tree of figure 1. Each heading includes the coordinates of the architecture it describes.

A tremendous variety of topologies is possible for SIMD machines. In practice, though, a few simple topologies are used by most SIMD architectures. The simplest, 1D (1-dimensional), is a property of SIMD architectures. Although it will not be mentioned in their descriptions, all the other topologies contain it in addition to their advertised features. A 1D topology simply labels each of n PEs with a unique integer $0 < x \leq n$. PE x has two neighbors, $x - 1$ and $x + 1$. Boundary conditions can be defined so PEs 0 and $n - 1$ are neighbors (forming a ring), or so their missing neighbors (PEs -1 and n) always provide null values (forming a line segment). Since these architectures are equivalent, they will not be distinguished.

The most common topology is 2D, which labels each PE with an ordered pair (x, y) such that $0 < x < X$, $0 < y < Y$, and $n = XY$. Each PE has four or eight neighbors, differing by plus or minus one in one or both dimensions. Boundary conditions can be defined to provide wrap-around (forming a torus), or null boundary values (forming a rectangular sheet). The architectures using all the topologies allowed by these choices are equivalent, so they will not be distinguished. The remaining topologies will be discussed as necessary with the architectures using them. These include Cube-Connected Cycles, Arbitrary, and Complete graphs.

Oldfield/Williams/Wiseman/Brûlé (1D, 0, 0, 0, 2, 0, 2)—J. V. Oldfield, R. D. Williams, N. E. Wiseman, and M. R. Brûlé propose a CAM (Content Addressable Memory) with sufficient processing power at each row to qualify as a SIMD architecture (Ref. 26). (Simulation of arithmetic operations and the enable-bit is rather laborious, but possible with a constant number of operations.) There is no communication between PEs, but the 1D topology provides row addresses. There is no local addressing or parallel I/O.

Pixel-Planes 4 (2D, 0, 0, 0, 0, 0, 0)—Pixel-Planes 4 (Refs. 15, 14, 12) is designed for high-performance interactive graphics applications. It has a simple 2D topology. There is no communication between PEs, but the PE coordinates (x, y) are used to compute bilinear expressions of the form $ax + by + c$ at each PE (for scalar floating-point values a, b, and c). Although there is special hardware to evaluate these expressions quickly, they can be computed in constant time without it. These expressions can be used to display polygons and spheres very quickly. There is no local addressing, global logical-or, parallel I/O, or PE to host I/O. However, images can be displayed on a video monitor, with each PE providing the data for one pixel of the image.

Video display of data in most architectures is done by parallel output to a frame buffer. The fact that data can be seen, but not otherwise externally accessed due to the absence of I/O, is a minor anomaly of Pixel-Planes 4. Because it cannot influence algorithm selection, there is no need to recognize it in the taxonomy.

Pixel-Planes 5 (2D, 0, 0, 0, 1, 1, 0)—Pixel-Planes 5 (Refs. 17, 12) is designed to provide greater speed and flexibility in order to interactively display more complex and realistic images. With regard to the taxonomy, it differs architecturally from Pixel-Planes 4 only in providing global logical-or and parallel I/O.

However, it has hardware support for biquadratic expressions in x and y, in addition to bilinear expressions. It also has a MIMD host. Both of these differences provide significant constant-bounded speedups. In addition, multiple sets of PEs can be combined in a single system. A program may choose to treat them as separate machines controlled by different processes in the host, or as a single large machine controlled by a single logical process. This is similar to the partitioning allowed by the Connection Machine.

Nickolls/Cole (2D, 2, 1, 0, 0, 1, 1)—P. M. Nickolls and T. W. Cole (Ref. 25) present a fault-tolerant 2D processor array for image synthesis. It has a 2D topology, with globally selected neighbor communication. It does not provide local memory addressing or global logical-or. It also provides parallel I/O and allows the host to obtain data from certain PEs at the edge of the PE array.

The distinguishing feature of this machine is not visible architectually. It is a programmable interconnection network that allows defective PEs and network connections to be configured out of the machine by deleting rows or columns containing the defective hardware.

MPP (2D, 2, 1, 0, 1, 1, 2)—The MPP (Massively Parallel Processor) (Ref. 29) has a 2D topology and allows each PE to communicate with a locally chosen neighbor. There is only global memory addressing. Global logical-or and parallel I/O are provided, and the host can obtain data from any PE.

DAP (2D, 2, 1, 0, 1, 1, 2)—The Active Memory Technology DAP (Distributed Array Processor) (Ref. 27) — formerly the ICL DAP — architecture appears identical to that of the MPP, at the level under discussion. (However, I have not been able to verify support for global logical-or.)

Illiac IV (2D, 2, 1, 1, 0, 1, 2)—The Illiac IV (Ref. 19) is an early SIMD architecture. Its 2D topology provides communication between each PE and its immediate neighbors, with local neighbor selection. The PEs have local addressing of their memories. Global logical-or is not provided. There is support for parallel I/O, and PE to host I/O from any PE.

BLITZEN (2D, 2, 1, 1, 1, 1, 2)—BLITZEN (Refs. 6, 11, 7) builds on many ideas from the MPP. Its architecture differs primarily in providing local addressing of PE memory. The architecture is almost identical, at this level, to that of the Illiac IV, differing only in supporting global logical-or.

BVM (CCC, 2, 1, 0, 0, 1, 1)—The BVM (Boolean Vector Machine) (Ref. 38) arranges PEs in a CCC (Cube-Connected Cycles) network (Ref. 30). Each PE can communicate with its choice of its three neighbor PEs. Only global memory addressing is provided. Global logical-or is not provided. Parallel I/O is supported, and the host can read data directly from a single distinguished PE.

GF11 (Arbitrary Permutation, 1, 0, 1, 1, 1, 2)—The GF11 (designed to achieve 11 GFLOPS) (Refs. 5, 4) can provide multiple arbitrary permutations for inter-PE communication. Each permutation is defined by a directed graph which specifies the PE from which each PE receives data, with exactly one PE receiving data from each PE. A particular permutation is globally selected for each communication operation between PEs.

Local addressing, global logical-or, parallel I/O, and arbitrary PE to host I/O are all supported.

BSP (Complete, 3, 0, 1, 0, 1, 2)—The BSP (Burroughs Scientific Processor) (Ref. 20, pp. 326-327, 410-422) architecture provides a complete interconnection graph, and allows each PE to determine locally with which neighbor to communicate, and which memory location to use. Since the complete graph makes neighbors of every pair of PEs, this provides completely arbitrary locally controlled inter-PE communication. Collision resolution is by serialization.

Local addressing, parallel I/O, and arbitrary PE to host I/O are all supported. Global logical-or is not.

As discussed above, although the BSP's memory is physically global, its architecture is fully equivalent to the description just given.

CM (Complete, 3, 1, 1, 1, 1, 2)—The Thinking Machines CM (Connection Machine) (Refs. 18, 10, 1) architecture provides a complete interconnection graph, and allows each PE to determine locally with which neighbor to communicate, and which memory location to use. Since the complete graph makes neighbors of every pair of PEs, this provides completely arbitrary locally controlled inter-PE communication. Collision resolution can be by serialization or combination.

Local addressing, global logical-or, parallel I/O, and arbitrary PE to host I/O are all supported.

There is a discrepancy between the CM's architecture, which provides a complete graph connecting PEs, and its hardware, which provides a hypercube (also known as a binary n-cube). This is a result of its system software and the definitions given earlier in this paper. As previously discussed, those definitions require a machine's architecture to be equivalent to the lowest-level publically documented programming interface. For the CM, that interface is currently Paris (Parallel Instruction Set) (Ref. 1). Paris's operations provide the communication system described above, but they are currently implemented by a physical hypercube with routing hardware. Paris operations can take time proportional to the number of PEs, so the architecture and hardware are not equivalent.

Evaluating The Taxonomy

It is probably not possible to prove that a taxonomy of SIMD architectures is complete, in the sense of adequately classifying all possible architectures that will ever be imagined. A more reasonable test of such a taxonomy is twofold:

- Does it adequately classify each SIMD architecture in the literature?
- Does it adequately classify every SIMD architecture which could be formed by taking different combinations of features from SIMD architectures in the literature?

The previous paragraphs have begun the work of showing that the proposed taxonomy satisfies the first of these criteria.

The nature of the proposed taxonomy makes the second criterion trivial to establish, once the first has been established. The taxonomy defines a multi-dimensional orthogonal space without holes, with a one-to-one and onto relation between dimensions and architectural features. This ensures that any combination of features corresponds to a single defined point in the architectural space.

EXISTING SIMD LANGUAGES

The research reported in this paper is primarily concerned with procedural languages, with a level of abstraction similar to C, C++, Pascal, or Fortran. Languages of this type both allow and require the programmer to express an algorithm unambiguously. Except for eliminating obviously redundant operations arising from the way an operation is expressed, the compiler for such a language is not involved in algorithm selection.

Some other families of languages allow the programmer to express the computation in a less algorithmic form, leaving the language implementation more latitude in choosing an exact algorithm. Some claim that the relative algorithm independence of the program allows greater portability among diverse parallel architectures. This is most often claimed with regard to modest parallelism on MIMD (multiple-instruction multiple-data) architectures. However, the way the problem is stated by the programmer can have a perhaps subtle but nevertheless profound effect on the algorithm ultimately used. In my opinion, this effect often ties such programs to a particular architecture as effectively as a procedural program expressing the same algorithm. I am not aware of any work on the use of non-procedural languages to programm SIMD architectures. Non-procedural languages will not be discussed further.

Survey of SIMD Languages

A careful search of the literature has found no SIMD programming languages satisfying the definition of optimal portability. Most existing languages for SIMD computers include implicit architectural assumptions. These limit them to some subset of the architectural space defined in the previous section. Some languages are not portable at all. To my knowledge, only one language, Fortran 8x, has been implemented on more than one SIMD machine. However, none is a complete implementation, and it is not clear how similar the subsets are. In the brief survey of SIMD languages below, languages other than Fortran 8x are grouped by machines. Very low-level languages are not considered, leaving no languages to discuss for some machines.

Illiac IV Languages—Three main languages were developed for the Illiac IV: GLYPNIR (Algol-like), CFD (Fortran-based), and IVTRAN (Fortran-based). (Ref. 19) All require the programmer to use and understand low-level hardware features and limitations. They are not true high-level languages. A more portable Pascal-based language called Actus (Ref. 28) was also developed. Actus is limited by its assumption of 2D grid communication.

MPP Language—The MPP's implementation of Parallel Pascal also fails to insulate programmers from hardware details, contrary to the language definition. Even as defined, Parallel Pascal is suitable only for architectures with a 2-dimensional rectangular inter-PE communication network. (Ref. 29)

CM Languages—Likewise, C* and Connection Machine Lisp, two admirably well-designed high-level languages for the CM, assume the presence of the CM's powerful, expensive, and almost unique support of arbitrary inter-PE communication. (Refs. 10, 31, 33)

BVM Language—BVL-0 (Boolean Vector Language 0) (Refs. 36, 37) is a C-like language for the BVM. It was designed to be the only language for the BVM, so it includes some very low-level machine-specific features. It assumes the presence of a CCC network, and does not provide for features not present in the BVM, like local addressing. Although it could be adapted for use on other architectures with a constant number of adjacent PEs, programs written to use the BVM's CCC network would have to be rewritten.

BSP Language—The BSP Fortran Vectorizer (Ref. 20, pp. 417-422) combines some automatic vectorization of ordinary Fortran with some vector-oriented language extensions. Some of these extensions assume the presence of the BSP's arbitrary communication.

Fortran 8x—A language consisting of Fortran 77 with some VAX extensions and some proposed Fortran 8x array extensions and a few machine-specific features was proposed in 1984 (Ref. 24), but not implemented (Ref. 3). More recently, a subset of Fortran 77, with proposed Fortran 8x array extensions (including some "removed extensions"), has been implemented for the CM (Ref. 3). FORTRAN-PLUS for the DAP 500 is an implementation of Fortran 77, minus I/O facilities, plus some proposed Fortran 8x array extensions (Refs. 27, 2). It is not yet clear how compatible these implementations are.

The proposed Fortran 8x standard (Ref. 35) is the most portable language yet implemented for SIMD architectures. Although it is not optimally portable, its "removed extensions" are a step in that direction because they can be implemented on those architectures that support them efficiently. They include vector-valued array subscripts, which require arbitrary communication. Still, Fortran 8x requires communication and uses 2D grid communication heavily, so it cannot be implemented on all SIMD architectures.

Existing Languages Fail

Each of these languages contains embedded assumptions about the architecture or architectures on which programs will run, violating the first part of the definition of optimal portability. The discussion of each language commented on these assumptions. Every language discussed allowed the use of one or more features not present in all architectures, and most failed to allow the use of some feature present in some architecture. Therefore, they all failed to satisfy the second or third part of the definition of optimal portability.

AN OPTIMALLY PORTABLE LANGUAGE

A programming model is a complete description of the visible features and behavior of a computer system, as seen by a program. One reason existing SIMD languages are not optimally portable is each one provides only a single programming model, reflecting a fixed set of architectural features and assumptions. The second programming model provided by Fortran 8x's "removed extensions" is a small step away from this problem, but Fortran 8x still embodies many architectural assumptions.

An optimally portable SIMD language must support a family of programming models corresponding to the architectures defined by a taxonomy like the one proposed above. Each model is specified by the coordinates of its point in architectural space. Thus, each model embodies the architectural requirements of the algorithms expressed in that model.

Porta-SIMD is a new language which will provide these programming models. Its design and prototype implementation are being carried out to demonstrate the feasibility and power of optimally portable SIMD languages. It is not intended to be the only or ultimate such language, but to stimulate the development and use of optimally portable languages. For this reason, some compromises have been made in aesthetic details of the language, and in performance, in order to proceed in a timely manner with limited resources.

These considerations contributed to the choice of C++ (Ref. 34) as the base language for Porta-SIMD. There was no need nor time to invent new syntax and semantics for the scalar and sequential sections of SIMD programs, and much to be gained by using a language with which programmers were already familiar. SIMD parallel datatypes and operations can be expressed as classes and overloaded operators in C++, extending the language cleanly without modifying the compiler. This would not have been true with Fortran, C, or Pascal.

Porta-SIMD defines a set of classes, one per data type, for each programming model, and a model for each point in the architectural space defined by the taxonomy proposed above. The models are derived (using C++ inheritance) from the base model, which implements the "least common denominator" SIMD architecture (1D, 0, 0, 0, 0, 0, 0). C++'s coming multiple inheritance will be used to derive an arbitrary model from the base model and an additional model for each architectural dimension along which the arbitrary model has features above the base model. This will prevent the implementation effort from exploding combinatorially with the size of architectural space.

```
/* Define programming model: (2D,0,0,0,0,0,0) */
#include <simd_int_2d.h>
simd_mach_2d mach;

/* square accepts the upper left and lower right
 * corners of a square.  Returns 1 in each PE
 * inside the square, 0 in each PE outside.
 */
simd_int_2d square(int x1, int y1, int x2, int y2)
{
  simd_int_2d inside(mach, 1);
  simd_int_2d x(mach, 16), y(mach, 16);
  inside = 1;
  x.coord_x();
  y.coord_y();
  inside &= (x > x1);
  inside &= (y > y1);
  inside &= (x < x2);
  inside &= (y < y2);
  return(inside);
}

main()
{
  display(-square(2,6,24,57));
}
```

Figure 2. Example Porta-SIMD program.

Parallel expressions are evaluated at each active PE according to the normal C++ rules.

A parallel language needs parallel control structures, as well as parallel data types. It is sufficient to extend the semantics of the `if` statement to allow a parallel value in the test expression. An element of this value is used by each PE to to determine whether to execute the body of the `if` or the `else` clause following the test. Unfortunately, C++ does not provide a means to extend the semantics of control structures, like it does for data types. This semantic extension could be accomplished by a conceptually simple Porta-SIMD to C++ pre-processor which replaced parallel `if` statements with small blocks of code to enable and disable PEs appropriately. Unfortunately, writing such a pre-processor (or deriving one by modifying a C++ compiler) is a difficult and time-consuming task in practice. For now, a few macros are used to express parallel `if` statements, instead. For example, if p is a parallel variable,

```
if (p)
          a;
else
          b;
```

is instead written as

```
IF (p)
          a;
ELSE
          b;
ENDIF
```

A more detailed language description is beyond the scope of this paper. A sample program is shown in figure 2.

Choosing to implement Porta-SIMD primarily as C++ classes has both welcome and unwelcome consequences. The primary benefit is avoiding the need to write a compiler. The amount of work this saves cannot be overemphasized. Another benefit is that the Porta-SIMD prototype is itself very easy to port: C++ is widely available, and the prototype has been written in a coding style which carefully separates machine-independent from machine-dependent code. The primary disadvantage is that the evaluation of parallel expressions proceeds operator by operator, without any overview of the expression. This is because the code implementing each parallel operator has no way to know anything about its place in the expression. The result is that extraneous temporary values and redundant copies are sometimes necessary, reducing execution efficiency. Although this would probably be unacceptable in a production-quality language implementation, it is acceptably small for the current purposes. It is certainly possible to write an optimizing compiler for Porta-SIMD, but this is well beyond the scope of the current research.

Initial development was done on Pixel-Planes 4, a 256K PE machine in regular use at UNC. The base model (1D, 0, 0, 0, 0, 0, 0) was ported to a 16K PE CM-2 in five days, including the time required to learn Paris. This was done in the ACRF (Advanced Computing Research Facility) at Argonne National Labs. The Pixel-Planes 4 model (2D, 0, 0, 0, 0, 0, 0) is now running on both Pixel-Planes 4 and the CM. Integers of all sizes are supported. However, floating point types have been deferred while effort focuses on the central architectural and language design issues. Other models are in various stages of development. A port to the Pixel-Planes 5 simulator is planned for the near future. No performance tuning or detailed measurements have been attempted, but this early prototype obviously provides lots of room for improvement. A few brave early users are already providing valuable and encouraging feedback.

CONCLUSIONS

The extraordinary architectural diversity of SIMD computers is too important to algorithm selection to completely hide from programmers. Optimal portability is a new concept for managing this architectural diversity. It provides specific criteria for identifying the architectural features a programmer needs to see. It allows the programmer to precisely specify the portability of each program. This lets the programmer judge the proper tradeoff between acheiving broad portability and taking full advantage of a particular architecture. Existing languages usurp this decision with predetermined architectural assumptions.

Porta-SIMD is being implemented to demonstrate the power and feasibility of optimally portable languages. It takes advantage of C++ classes and operator overloading to reduce the implementation effort. Although only a few programming models have been implemented so far, Porta-SIMD is already running on Pixel-Planes 4 and a CM-2. This is probably the first language to be implemented identically on more than one SIMD computer.

Although optimal portability has been applied here to SIMD architectures, it is potentially valuable for any diverse but related class of architectures.

ACKNOWLEDGEMENTS

Profs. Frederick Brooks and Henry Fuchs have given me valuable advice, support, and encouragement. Greg Turk has been a valuable sounding board for ideas and problems. He and Brice Tebbs were willing to use Porta-SIMD very early, and have given me valuable feedback. It is a privilege to be part of the Pixel-Planes team. I appreciate the people and facilities of the ACRF, especially Rick Stevens, who helped me get started on their CM.

REFERENCES

1. *Connection Machine Parallel Instruction Set (Paris): The C Interface (Version 4.0).* Thinking Machines Corporation, Cambridge, MA, 1987.

2. *DAP 500 Introduction to FORTRAN-PLUS Programming.* Active Memory Technology Limited, Reading, UK, 1987.

3. Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. *SIGPLAN Notices*, 23(9):42–56, September 1988. (Proceedings of the ACM/SIGPLAN PPEALS 1988).

4. John Beetem, Monty Denneau, and Don Weingarten. GF11. *Journal of Statistical Physics*, 43(5/6), June 1986.

5. John Beetem, Monty Denneau, and Don Weingarten. The GF11 supercomputer. In *IEEE Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 108–115, June 1985.

6. Donald W. Blevins, Edward W. Davis, and John H. Reif. *Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor*. September 1987.

7. Donald W. Blevins and R. A. Heaton. The BLITZEN PE array chip feature set. In *Second Symposium on the Frontiers of Massively Parallel Computation*, October 1988.

8. K. Mani Chandy and Jayadev Misra. Architecture independent programming. In *Third International Conference on Supercomputing, Vol. 3*, pages 345–351, International Supercomputing Institute, Inc., 1988.

9. D Chin, J Passe, F Bernard, H Taylor, and S. Knight. The Princeton Engine: a real-time video system simulator. *ICCE*, May 1988.

10. Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*. Technical Report HA87-4, Thinking Machines Corporation, April 1987.

11. Edward W. Davis and John H. Reif. Architecture and operation of the BLITZEN processing element. In *Third International Conference on Supercomputing, Vol. 3*, pages 128–137, International Supercomputing Institute, Inc., 1988.

12. John Eyles, John Austin, Henry Fuchs, Trey Greer, and John Poulton. Pixel-Planes 4: a summary. In *Proceedings of Eurographics '87 Second Workshop on Graphics Hardware*, 1987.

13. T. J. Fountain. A survey of bit-serial array processor circuits. In M. J. B Duff, editor, *Computing Structures for Image Processing*, chapter 1, Academic Press, Inc., Orlando, FL, 1983.

14. Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. *Computer Graphics*, 19(3):111–120, July 1985. (Proceedings of SIGGRAPH '85).

15. Henry Fuchs and John Poulton. Pixel-Planes: a VLSI-oriented design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

16. F. A. Gerritsen. A comparison of the CLIP4, DAP, and MPP processor-array implementations. In M. J. B Duff, editor, *Computing Structures for Image Processing*, chapter 2, Academic Press, Inc., Orlando, FL, 1983.

17. Jack Goldfeather, Jeff P. Hultquist, and Henry Fuchs. Fast constructive solid geometry display in the Pixel-Powers graphics system. *Computer Graphics*, 20(4):107–116, July 1986. (Proceedings of SIGGRAPH '86).

18. W. Daniel Hillis. *The Connection Machine. MIT Press Series in Artificial Intelligence*, The MIT Press, Cambridge, MA, 1985.

19. R. Michael Hord. *The Illiac IV: The First Supercomputer*. Computer Science Press, Inc., Rockville, MD, 1982.

20. Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, New York, 1984.

21. Leah H. Jamieson. Features of parallel algorithms. In *Second International Conference on Supercomputing, Vol. 1*, pages 476–478, International Supercomputing Institute, Inc., 1987.

22. Alan H. Karp. Programming for parallelism. *Computer*, 43–56, May 1987.

23. A. Krikelis and R. M. Lea. Low-level vision tasks using parallel string architectures. In *Parallel Processing for Computer Vision and Display*, January 1988.

24. *A FORTRAN Compiler for the Massively Parallel Processor*. Massachusetts Computer Associates, Inc., February 1984. CADD-8402-2101.

25. P. M. Nickolls and T. W. Cole. A fault-tolerant 2-d processor array for image analysis. In *Parallel Processing for Computer Vision and Display*, January 1988.

26. J. V. Oldfield, R. D. Williams, N. E. Wiseman, and M. R. Brûlé. Content-addressable memories for quadtree-based images. In *Proceedings of Eurographics '88 Third Workshop on Graphics Hardware*, 1988.

27. D. Parkinson, D. J. Hunt, and K. S. MacQueen. The AMT DAP 500. In *Spring COMPCON 88: digest of papers*, pages 196–199, The Computer Society of the IEEE, IEEE Computer Society Press, February 1988.

28. R. H. Perrott. A language for array and vector processors. *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, October 1975.

29. J. L. Potter, editor. *The Massively Parallel Processor. MIT Press Series in Scientific Computation*, The MIT Press, Cambridge, MA, 1985.

30. Franco P. Preparata and Jean Vuillemin. The Cube-Connected Cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

31. John R. Rose and Guy L. Steele Jr. *C*: An Extended C Language for Data Parallel Programming*. Technical Report PL87-5, Thinking Machines Corporation, April 1987.

32. Charles L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

33. Guy L. Steele Jr. and W. Daniel Hillis. *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing*. Technical Report 86.16, Thinking Machines Corporation, May 1986.

34. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986.

35. Accredited Standards Committee X3 - Information Processing Systems Technical Committee X3J3 - Fortran. *X3.9-198x: Draft Proposed Revised American National Standard Programming Language Fortran (Version 104)*. American National Standards Institute, April 1987.

36. Sherry J. Tomboulian, Mary Mace, and Robert A. Wagner. Language report and description for BVL-0. April 1985. Unpublished paper.

37. Russell R. Tuck, III. *Issues in the Design of an Optimizing Code Generator for BVL-0*. Master's thesis, Duke University, Durham, NC, 1987.

38. Robert A. Wagner. The Boolean Vector Machine (BVM). In *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 59–66, 1983.