Porta-SIMD: An Optimally Portable SIMD Programming Language

Duke CS-1990-12 *UNC CS TR90-021*

May 1990

Russ Tuck

Duke University
Deparment of Computer Science
Durham, NC 27706

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Text (without appendix) of a Ph.D. dissertation submitted to Duke University. The research was performed at UNC.

PORTA-SIMD: AN OPTIMALLY PORTABLE SIMD PROGRAMMING LANGUAGE

by

Russell Raymond Tuck, III

Department of Computer Science
Duke University

Date: April 26, 1990

Approved:

[illegible]

Frederick P. Brooks, Jr., Supervisor

[illegible]

John A. Board

[illegible]

Henry Fuchs

Merrell L. Patrick

Merrell L. Patrick

Jonathan B Rosenberg

Jonathan B. Rosenberg

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

1990

Abstract

Existing programming languages contain architectural assumptions which limit their portability. I submit optimal portability, a new concept which solves this language design problem. Optimal portability makes it possible to design languages which are portable across various sets of diverse architectures. SIMD (Single-Instruction stream, Multiple-Data stream) computers represent an important and very diverse set of architectures for which to demonstrate optimal portability. Porta-SIMD (pronounced "porta-simm'd") is the first optimally portable language for SIMD computers. It was designed and implemented to demonstrate that optimal portability is a useful and achievable standard for language design.

An optimally portable language allows each program to specify the architectural features it requires. The language then enables the compiled program to exploit exactly those features, and to run on all architectures that provide them. An architecture's features are those it can implement with a constant-bounded number of operations. This definition of optimal portability ensures reasonable execution efficiency, and identifies architectural differences relevant to algorithm selection.

An optimally portable language for a set of architectures must accommodate all the features found in the members of that set. There was no suitable taxonomy to identify the features of SIMD architectures. Therefore, the taxonomy created and used in the design of Porta-SIMD is presented.

Porta-SIMD is an optimally portable, full-featured, SIMD language. It provides dynamic allocation of parallel data with dynamically determined sizes. Generic subroutines which operate on any size of data may also be written in Porta-SIMD. Some important commercial SIMD languages do not provide these features.

A prototype implementation of Porta-SIMD has been developed as a set of `#include` files and libraries used with an ordinary C++ compiler. This approach has allowed more rapid prototyping and language experimentation than a custom compiler would have, but modestly constrained the language's syntax. The result is a very portable but only moderately efficient implementation. Porta-SIMD has been implemented for the Connection Machine 2, for Pixel-Planes 4 and 5, and for ordinary sequential machines.

Optimal portability is an important new concept for developing portable languages which can handle architectural diversity. Porta-SIMD demonstrates its usefulness with SIMD computers.

Acknowledgements

I am most thankful to God, who has made possible what I could not have done alone. He enabled me to finish work which looked like it could drag on forever, and to do it in time for an impossible looking graduation deadline. He taught me a lot about faith, trust, prayer, and His peace in the process. He has given me a wonderful wife, Debbi, who loves me and makes life fun. Most of all, God gave His son Jesus to die for my failures and give me new life with meaning and hope.

I appreciate Debbi's unending patience, prayers, sacrifice, and encouragement. These have been especially important during my final push to finish, during which she has been busy finishing her own graduate degree. Our family, our church and some special friends have joined in with encouragement and prayers, for which I am very grateful. Dad's quick mid-day phone calls from California several times over the last few weeks were especially helpful.

Dr. Frederick P. Brooks, Jr. has been the consummate advisor. He has provided critically important insight, advice, and perspective. He has regularly and dependably scheduled time for our meetings, despite the many demands on his time, and that has been very important to my steady progress. I am grateful to Dr. Brooks for accepting me as his student and providing me with assured grant support, guidance, and freedom, even when all I knew about my research goals was that I wanted to improve the programming of SIMD computers. I appreciate his deep Christian faith, and have enjoyed participating in Wednesday lunch Bible studies with him.

I appreciate my committee for their encouragement and sound advice. Dr. Henry Fuchs has supported my work financially as part of the Pixel-Planes project since I began working on Porta-SIMD, and has been very supportive personally as well. Dr. Merrell Patrick also served on my M.S. committee, and has shown a personal interest in my success that was especially helpful during my transition into doctoral research. Dr. Jothy Rosenberg's ideas, interest, and personality have made him a very valuable source of encouraging advice and positive suggestions. Dr. John Board has provided sound comments and helpful references.

I appreciate Dr. Jan F. Prins' service on my committee as an *ex officio* member, doing significant work without official recognition. He has often been the most accessible member of my committee, and has served as a valuable sounding board for ideas and dilemmas.

Greg Turk, a fellow graduate student and member of the Pixel-Planes team, has been valuable as a pioneering Porta-SIMD user and for his willingness to discuss and comment on how Porta-SIMD should work. I appreciate his willingness and that of Tim Cullip for me to use their programs as examples and include them in this dissertation.

I want to thank Michael Tiemann for writing G++ (the GNU C++ compiler), for making it freely available, and for providing timely fixes to compiler bugs as they were reported.

I appreciate the encouragement and friendship of many members of the Duke and UNC Computer Science Departments, including especially my long-time office mates at Duke, Jack Briner and Mark Jones, and the entire Pixel-Planes team at UNC.

I appreciate the time Carlton Brown and Debbi Tuck spent patiently and carefully reading drafts of this document to point out and suggest corrections for my writing errors.

I appreciate the time Herb Taylor of the David Sarnoff Research Center spent helping me understand details of the Princeton Engine, and also his friendship and encouragement.

Several institutions have been important to my research. The Departments of Computer Science at the University of North Carolina at Chapel Hill and Duke University have provided critically important computing resources and office space. My research at UNC has been supported by the Pixel-Planes Project, Drs. 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 first by the Advanced Computing Research Facility (ACRF) at Argonne National Laboratories, under grants NSF-ASC-8808327 and DOE-W-31-109-ENG-38, and more recently by the Connection Machine Network Server (CMNS) Pilot Facility at Thinking Machines Corporation under DARPA contract DACA76-88-C-0012.

Contents

List of Tables

List of Figures

Chapter 1

Introduction, Thesis, and Overview

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) parallel computers has kept them from fully benefiting from such languages. Each existing SIMD language assumes some particular set of features is available, and is therefore suitable for programming only some subset of SIMD machines. Architectural differences such as communication and local addressing are too important for algorithm selection to be hidden from the programmer in this way. They are too powerful to ignore when available, but too expensive to simulate when not supported in hardware.

Optimal portability is a new language design concept which can bring the benefits of portable high-level languages to architecturally diverse families of computers, including SIMD computers. Here is an informal definition of optimal portability: given a set of architectures, and a taxonomy for those architectures which can be expressed in a tree structure, a language is optimally portable for the set of architectures if: (1) it requires each program to specify at compile time a subtree of the taxonomy as the set of architectures on which it will run; (2) it allows the program to take full advantage of the features common to all the nodes of that subtree; and (3) it prevents the program from using any features not common to all the nodes of that subtree. The full definition adds a key requirement: the taxonomy must consider two nodes, or a node and a computer, equivalent if and only if each can simulate the other with a constant-bounded number of its own operations. This is called the *constant-bounded simulation criterion* within this work.

The basis for optimal portability is recognizing that each algorithm has some inherent range of portability implicitly defined by the operations (or architectural features) it uses. An optimally portable language allows every algorithm appropriate for its set of architectures to be expressed naturally, and to execute on all architectures which support the set of operations used by the algorithm. An optimally portable language provides some consistency checking: it generates an error message if the program uses operations not common to all nodes in the specified subtree of the taxonomy. Finally, the constant-bounded simulation criterion ensures reasonable execution efficiency. In short, an optimally portable language lets the programmer use all the features shared by any specified set of target architectures, while being protected from using features they do not share.

By specifying a broad or narrow set of target architectures, the programmer can select just the right tradeoff between universal portability with limited features and limited portability with more powerful features. The language thereby provides the optimal degree of portability for each program. An additional benefit is that the programmer may use a single language for many architectures instead of having to learn many different languages.

I have developed a taxonomy of SIMD architectures which uses the constant-bounded simulation criterion. Using it, I designed Porta-SIMD (pronounced "porta-simm'd"), the first optimally portable SIMD programming language. Porta-SIMD provides a set of types corresponding to each point in the taxonomy. These types support exactly the operations appropriate to that point. By declaring the data types of each algorithm's variables, the programmer specifies which operations the algorithm may use. The set of all data types used in a program's declarations determines the program's portability. The program will run on any architecture which supports all the architectural features implied by the set of data types.

Porta-SIMD has been implemented on Pixel-Planes 4 and 5 and the Connection Machine, as well as on a sequential computer simulating parallel architectures.

I am prepared to defend the following thesis statements:

- Optimal portability is an important new concept for handling architectural diversity in programming language design, and is particularly important for SIMD architectures.
- Porta-SIMD is the first optimally portable programming language for SIMD computers. It demonstrates that optimal portability is an achievable goal for SIMD languages.

My argument that these statements are correct has three main parts: analysis, design, and implementation.

The analysis portion of this dissertation contains three chapters. Chapter 2, "Optimal Portability", presents a precise definition of optimal portability, and discusses issues involved in formulating this definition.

None of the existing SIMD taxonomies used the constant-bounded simulation criterion, so I developed one that does. Chapter 3 presents this new taxonomy and discusses previous SIMD taxonomies. It demonstrates the new taxonomy's completeness by using it to characterize a large number of published SIMD architectures. While developing this taxonomy was a necessary step in applying the concept of optimal portability to SIMD language design, the taxonomy is a useful contribution to knowledge in its own right. It is more complete in coverage and comprehensive in scope than previous work, and benefits from the well-defined and defensible level of detail provided by optimal portability's constant-operation simulation criterion.

Despite the wide variety of existing SIMD languages, none are optimally portable. Chapter 4, "Portability of Existing SIMD Languages," surveys existing SIMD languages and shows how each fails to satisfy the definition of optimal portability. The principal way these languages fail is that each language provides only a single set of architectural features for all programs to use. As a result, each language can only be implemented on architectures which support all the features it requires. And if the architecture provides additional features, the language provides no way to take advantage of them. The language determines the degree of portability instead of the programmer, which is the opposite of optimal portability.

The design portion of the dissertation consists of chapter 5, "Porta-SIMD Language Design." It describes the first optimally portable SIMD programming language. Porta-SIMD was designed to show how the concept of optimal portability can be applied to language design. For convenience, Porta-SIMD is an extension of C++. Porta-SIMD provides parallel data types with an additional characteristic included in the type: the set of architectural features available for using and manipulating data of that type. Because these features are those defined by the taxonomy of chapter 3, this allows the program to specify exactly which features are used by each algorithm and each program. The implementation provides type checking to ensure that features not specified by an object's type are not used with that object.

The primary argument in support of the dissertation's thesis concludes with chapter 6, "Implementing an Optimally Portable Language". Just as the design of Porta-SIMD was undertaken to demonstrate the value and applicability of optimal portability, a prototype implementation of Porta-SIMD was done to demonstrate the completeness and usefulness of the language design.

This chapter describes the requirements an implementation of any optimally portable language must meet, then moves on to the specific prototype implementation of Porta-SIMD. Because Porta-SIMD's implementation is intended as a proof-of-concept tool, it has been done as a set of libraries and `#include` files for use with an unmodified C++ compiler. Despite some limitations, C++ was the right tool for this stage of the research, allowing successively more complete prototypes on a wide variety of platforms. Writing a full optimizing compiler for Porta-SIMD would have been inappropriate for this work, and impossible with the resources available.

Porta-SIMD has been implemented for the Connection Machine, Pixel-Planes 4, the Pixel-Planes 5 simulator (hardware was not yet available), and sequential computers simulating SIMD architectures. In addition to describing the Porta-SIMD implementation, chapter 6 presents some example Porta-SIMD programs. It goes on to discuss the performance of this multi-targeted implementation and potential future implementations, using both analytic methods and actual measurements based on the example programs. There is no reason to believe that a compiler for an optimally portable language cannot provide performance comparable to other languages which are not optimally portable but have equivalent features otherwise. The price of the portability appears to be paid entirely in compile time. However, using the *prototype* Porta-SIMD implementation, programs run up to 10 times slower than using machine-specific SIMD languages, and potentially use 50

Chapter 7, "Recommendations," follows the formal argument in support of the thesis. It presents opportunities for future research and my personal view of how the concept of optimal portability should influence existing and future SIMD languages. It begins with opportunities related specifically to the prototype implementation of Porta-SIMD, and the Porta-SIMD language design. This leads to a discussion of improved optimally portable languages, and other characteristics I consider important for SIMD languages. Finally, I discuss the interface between the host and the PEs. It can be improved through both hardware architecture and compiler technology.

Finally, chapter 8, "Conclusions," summarizes the results of this research. Optimal portability appears to be a clarifying concept for handling architectural diversity. Its application in Porta-SIMD provides a new mode of portability for SIMD languages, and demonstrates that such portability is achievable in practice. The requirements of optimal portability contributed to the development of a SIMD taxonomy in which equivalence is defined by constant-bounded simulation. This taxonomy consequently has many dimensions of equivalence-versus-difference.

I hope this work will have a strong influence on future SIMD languages, and that optimally portable languages will bring more, cheaper, and better software to all SIMD architectures.

Chapter 2

Optimal Portability

An optimally portable language allows a programmer to write code which is exactly as portable as the algorithm being coded. This is the optimal degree of portability: exactly what is needed for each situation. Defining this level of portability, and making it available in a SIMD programming language, is the key contribution of this dissertation.

In contrast to optimal portability, existing SIMD languages artificially restrict or extend portability. Some restrict portability by requiring specific architectural features (e.g., arbitrary global communication) to be present whether they are used or not. Some extend portability by not allowing the use of some architectural features. They thereby ban efficient execution of some algorithms. Most do both.

Of course, designing and implementing an optimally portable language requires a precise definition of optimal portability. This chapter presents and justifies that definition. A key feature of this definition is the strict lower bound it places on execution efficiency. This bound ensures that optimally portable languages provide useful portability, not just theoretical portability. Succeeding chapters demonstrate this definition's value by applying it to existing languages, and to the design and implementation of an example language.

Optimal portability is best defined using operations on sets of abstract architectures. Blaauw and Brooks define the *architecture* of a computer as "a minimal behavioral specification," and distinguish it from the implementation and realization. [BlaauwBroo90] Specifically, variable quantities such as the number of processors in a parallel computer and the amount of memory per processor are not specified by the architecture. They are details of the hardware realization. Within this work, I define an *abstract architecture* to be the set of data types and operations provided by a computer's architecture, without regard to how the data and operations are represented. Every data type is assumed to have a fixed size which is the same in all the processors of a system. Except where explicitly stated otherwise, I use *architecture* as a synonym for abstract architecture. Where I need to distinguish abstract architecture from architecture as defined by Blaauw and Brooks, I refer to the latter as *detailed architecture.* The operations and data types of an architecture are collectively referred to as the architecture's *features*, another synonym for abstract architecture.

Language definitions share many characteristics with architectures (which are themselves machine languages). Languages are implemented by compilers and interpreters. Dealing with an abstract (as opposed to a detailed) architecture gives the degree of portability found in most sequential languages, such as C and Fortran. These languages provide portability only between closely related architectures following the von Neumann model. [Dasgupta89, pp. 103–108] Note that languages do not impose limits on architectural variables. But the implementation of a language may fail to execute a program which is perfectly legal according to the language, if some resource (e.g., memory) is exhausted during its execution.

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 definition of intersection is more complicated, and introduces key elements in the overall definition of optimal portability. The point of the two-step construction of a set's intersection is to synthesize the most powerful architecture which can be simulated efficiently by all members 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. The first step in constructing a set's intersection expands each member architecture to include all the features it can simulate efficiently. Then the second step can recognize the features shared by all members, even if they were initially expressed differently. The two most important parts of the definition of intersection are the phrases "number of ...data elements and operations" and "constant." These phrases address the issue of efficiency.

First, the number of operations and data elements is measured, not time (in clock cycles or milliseconds) or space (in processor word size, bits of memory, logic elements, or silicon area). Measures of time and space apply to implementations and realizations, not architectures. Architectures have operations and data elements; simulation efficiency is measured by counting these. The dynamic number of operations executed is counted, not the static number written in an algorithm or the number of distinct types used. This corresponds to the implementation concept of execution time, rather than code size or instruction set size. Data elements are counted by the high-water mark, or the maximum number ever allocated simultaneously. In implementation terms, this would be the amount of memory required to execute the code.

The second phrase, "constant," introduces the constant-bounded simulation criterion which is used to distinguish architectures and their features. *Constant-bounded simulation* is the simulation by some architecture of one or more architectural features, using only a constant number of the architecture's operations and data elements. Any Turing-equivalent machine can simulate any architecture, but not always with useful performance. The constant-bounded criterion provides a tight bound on simulation overhead. It also fits well with intuitive notions of equivalent architectures, by making equivalence transitive. If architecture A_i can do a constant-bounded simulation of A_j, and A_j can likewise simulate A_k, then A_i can also do a constant-bounded simulation of A_k. Though the simulation of A_k by A_i may be a two-level process that multiplies the constants of the two levels, the product of any finite set of constant-bounded numbers is also constant-bounded. It will be shown in chapter 3 that the set of distinct features of SIMD architectures is finite.

A looser bound on simulation for defining equivalent architectures, such as a logarithmic or polynomial bound in the amount of memory or number of processors, would not allow equivalence to be transitive. A tighter bound, such as a limit on the constants allowed in constant-bounded simulation, would be arbitrary. It would also be meaningless in the face of large constant-factor differences in execution times for identical features on different implementations.

The result of the definition of a set of equivalent architectures is that any member of the set can simulate any other member, and the number of native operations they execute will be within a constant factor of each other.

An architecture *supports* a language feature (which may be an operation, statement type, data type, standard library function, or other construct) if and only if the language feature can be implemented with a constant number of the architecture's operations and data elements. A set of language features *provides access to* (or *provides*) an architectural feature if and only if it enables a program to express within a constant-bounded number of symbols all computations performable by a single use of the architectural feature. A program is *portable* across a set S of architectures if and only if it uses only language features supported by the shared architecture of S. The *target architecture* of a program is an architecture across which the program declares it is portable.

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 declare 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 language feature not supported by A_p.
- L provides all architectural features in A_p, and allows p to use any language feature supported by A_p.

By satisfying this definition, an optimally portable language gives p exactly the portability and power implied by its architectural assumptions, as embodied in its target architecture A_p. As a result, p is as portable as possible without changing its architectural assumptions. Of course, if p has specified a target architecture supporting architectural features it actually does not use, then p can be made more portable simply by specifying a target architecture which more precisely reflects the program's intrinsic portability. The 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. (But if p does not actually use all the features in its target architecture, then giving up the "use" of these unused features by changing the target architecture gives p its full intrinsic portability without otherwise changing the program.) In addition, p cannot use additional data types or operations without adding to A_p. If the augmented architecture were not equivalent to A_p, this would reduce p's portability by removing architectures from S_p. If it were equivalent, it would not increase the power available to p.

It is important that optimally portable languages are not allowed to provide language features not supported by the target architecture. Providing such features would require the compiler to generate code which uses arbitrary data elements and operations to simulate the unsupported features. That would destroy the constant-bounded execution guarantee for language features.

Prohibiting compilers from simulating data types and operations not supported by the target architecture helps ensure portability with useful performance, not just theoretical portability. This does not restrict the function of programs, since a program may simulate such data types and operations itself. The implementers of a language may even provide, as a convenience to programmers, an optional library to do this simulation.

The definitions given so far ensure constant-bounded execution overhead from language features down through abstract architectures. However, this performance guarantee is meaningless without a similar assurance concerning execution overhead in implementing the architecture. A particular computer must 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 publicly documented programming interface. This definition says that every computer implements a single architecture, and specifies how to determine the definition of that architecture.

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 publicly documented programming interface. With an additional layer of software to do automatic routing, the machine might also be described as providing communication between arbitrary pairs of PEs. The number of operations required to simulate arbitrary communication with this 2-D grid 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 publicly documented programming interface, the architecture would be considered by the above definitions to provide communication between arbitrary pairs of PEs.

For most sequential computers, the lowest-level publicly documented programming interface is assembly language. For some SIMD computers it is a library, or even a high-level language and its standard library. For example, on the Connection Machine the lowest-level publicly documented programming interface is Paris, a library callable from several sequential languages. On the MasPar MP-1, it is MPL (which is C with simple parallel extensions) and its standard library.

There are several reasons to define a machine's architecture by its lowest-level publicly documented programming interface, rather than by its hardware. A programmer has no access to the hardware except through this interface, so a language which provides features equivalent to this interface is neither hiding features from nor inventing features for the programmer. Hardware documentation is not always publicly available. When it is, 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, transparently to the programmer, a single architecture with different hardware designs. These identically programmed machines should be considered by languages to have the same architecture.

Although I believe this is the best available method for defining a computer's architecture, it leads to some apparent contradictions. For example, the CM-2 (Connection Machine, model 2) and MP-1 both define global reduction and parallel prefix computations as single operations. This is despite the fact that any implementation using pairwise operations (which they in fact use) must take $O(\log_2 n)$ steps when there are n PEs. Arbitrary global inter-PE communication is another example. Both the CM-2 and the MP-1 implement this operation with an iterative procedure, the number of iterations of which is not bounded by a constant with respect to the number of PEs. Even sequential machines have a slight non-constant element to their implementation; memory access with address decoding is theoretically logarithmic in the size of the address space (linear in the number of address bits). However, the architects and implementers of these machines, and those of all the languages designed and implemented for them, concur that these features should be considered part of the programming model. They are powerful abstractions, and can be implemented reasonably efficiently with a combination of hardware and firmware. Defining these machines' architectures by their lowest-level publicly documented programming interface captures this collective judgement, which further supports the use of this definition.

The definition of optimal portability allows a strong statement to be made about the efficiency of optimally portable languages: every language feature available on a particular computer is implementable with a constant number of operations and data elements at the computer's lowest-level publicly documented programming interface. This is as strong a statement of efficiency as any portable language can make. Anything stronger would require knowledge of implementations, which is beyond the scope of language definitions.

Chapter 3

A SIMD Taxonomy for Optimal Portability

While the architectural differences between sequential (not vector) computers are significant to cost and performance, it is very rare for them to influence algorithms in application programs. This is because purely sequential computers share the von Neumann model of architecture, and are equivalent by the definition in chapter 2. Therefore, sequential languages are optimally portable by that chapter's definition.

In contrast, the family of SIMD architectures is large and diverse. SIMD architectures aim to apply unmatched computational power to suitable problems. They do this by replicating the essential processing elements abundantly and distributing the data store among them, without also duplicating the instruction fetch and decode, flow control, and code store components. But choosing the precise set of features to replicate in each PE, and the network by which PEs exchange data, involves trade-offs between power and cost per PE. There is no single "right" set of trade-offs which is best for all applications, since applications vary in the benefit they derive from each feature. This is the source of the architectural diversity which makes optimally portable SIMD languages necessary and non-trivial.

A programming language can be optimally portable only for a specific set of architectures. That set must be well defined, so a definition of SIMD architectures is presented in section 3.1. The architectural features within the set must be identified and distinguished using the constant-bounded simulation criterion. I have developed a new SIMD taxonomy which does exactly that; it is presented in section 3.2. Section 3.3 demonstrates this taxonomy by using it to classify some important existing SIMD architectures. For the purpose of this classification, each SIMD computer is considered to implement the architecture defined by its lowest-level publicly documented programming interface. This is required by the definition of optimal portability. Finally, section 3.4 reviews some existing taxonomies and discusses why none of them is a suitable basis for designing an optimally portable language. They are discussed at the end of the chapter primarily so the vocabulary of architectures and architectural features developed in the rest of the chapter can be used in the discussion.

3.1 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 each 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 $0 \leq p \leq n-1$.

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 the instruction set.

Property 2 holds for every architecture which can, by any means, load into each PE a different member of a set of distinct values. To see this, consider the set of PE numbers as the data to be loaded. 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.

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 (cf., [HwangBrigg84, 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, section 3.3.18) is the only non-local memory architecture I know of. It is equivalent to a large subset of the CM (Connection Machine, section 3.3.20) architecture. (Both architectures are discussed briefly in section 3.3.) The BSP can simulate the CM's local memory model 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's global memory model by using its communication primitives to access memory, treating all the private memory as a single global memory space. Both are constant-bounded simulations, 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 memory and communication systems of the architectures is that the CM has more powerful mechanisms for resolving multiple simultaneous writes to a single memory location.

If any of these properties is not true of all SIMD architectures, then the taxonomy of the next section is considered to have an additional feature for each such property. Because all architectures currently classified by this taxonomy have all these features, they will not be mentioned further.

3.2 Taxonomy of SIMD Architectures

This section presents a new taxonomy of the diverse architectures that exist within the definition of SIMD just presented. The taxonomy's purpose is to provide a basis for designing optimally portable SIMD languages. Therefore, its criterion for distinguishing architectural features from one another is constant-bounded simulation.

Of course, some of the constant factors ignored by this criterion are important for other purposes. Designers of SIMD machines must decide not only which features to implement, but also how much hardware to devote to minimizing each constant. Constant factors can also be important to users choosing the best machine on which to run a program. But these constants reflect the performance of an implementation, not the suitability of an algorithm to an architecture. So this taxonomy appropriately ignores them.

The taxonomy presented here is reasonably comprehensive, but not exhaustive. No static SIMD taxonomy could remain exhaustive in any case, since new architectural features can always be invented. All that is needed is to propose implementing as a single operation some computation which cannot be implemented by existing architectures using constant-bounded simulation.

Communication networks are a good example of this. There is an infinite number of communication networks which are not equivalent (by the constant-bounded simulation criterion). A finite taxonomy cannot describe them all. Fortunately, the set of communication networks and other architectural features proposed in the literature is much more manageable. There are many common features and similarities, which makes it possible to describe most proposed architectures as combinations and variations of a few important architectural concepts.

The taxonomy presented here is intended to classify architectures which have been implemented or proposed in the literature. It is not intended to classify all conceivable architectures, or even all architectures obtainable by combining the features of proposed architectures in new ways. Where several architectures are sufficiently similar, I have sought to generalize the similar features and provide a unifying classification. For example, my description of cut-through communication is more general than connection autonomy as proposed by Maresca and Li [MarescaLi88, MarescaLi89]. I believe I am the first to identify the segmented communication bus architectures of the Princeton Engine (section 3.3.6 and ASP (Associative String Processor, section 3.3.7) as special 1-D cases of connection autonomy, and the first to integrate the communication capabilities of Unger's architecture (section 3.3.14) into a more general framework.

Some less common architectural features have not been used enough to suggest which possible generalizations are most useful. In particular, where pairs of features that might interact have never been proposed as part of the same architecture, I have not attempted to define what their interactions might be.

The taxonomy presented here classifies an architecture as a set of architectural features. A classification contains at most one feature chosen from each of 12 feature categories. The categories are grouped and named as follows. The groups are in order of decreasing significance for most algorithm selection tasks.

- Communication:
  - Labeling (number, N)
  - Communication (C)
  - Collision Resolution, Write (W)
  - Collision Resolution, Fetch (F)
  - Piped Communication (P)
  - Cut-Through Communication (T)
- Local Addressing:
  - Local Addressing (L)
- Reducing and Scanning:
  - Reduce (R)
  - Scan (S)
- Parallel I/O:
  - Input (I)
  - Output (O)

PE to Host I/O:
Get (G)

The items in parentheses identify the categories.

The sections that follow describe the naming of architectural features, and the features in each category. In some cases, a feature or set of features can perform a constant-bounded simulation of some other feature. The simulating feature or set of features is then said to *subsume* the simulated feature. This is mentioned where it is particularly important or common. Tables 3.1 (p. 15) and 3.2 (p. 16) summarize the features in each category.

3.2.1 Feature Names

Every feature has a name. A feature name consists of a number or an upper-case letter, optionally preceded by one or more lower-case letters. The upper-case letter identifies the category the feature is part of. Most categories have a unique letter which is the final (upper-case) letter of all its feature names. The single exception is "labeling"; it is the only category which uses numbers in feature names.

Several related features are sometimes named and described as variations of a single base feature. (E.g., adjacent communication has variations both with and without diagonal connections, wrap-around edges, and local communication.) In such cases, the base feature is named and described. Then each of a set of modifiers of that base feature is named and described. Rules are given describing the legal combinations of modifiers. Finally, each legal variation obtainable from the modifiers and combining rules is named.

The name of a variation always begins with the name of the base feature, minus the final letter. This is followed by the letters for all its modifiers, in the order the modifiers are described in the taxonomy. The final letter of the base feature completes the name of the derived feature.

In a few cases, there is no base feature because at least one modifier is required by all variations. Where this is true, a pseudo-base feature is given which has an asterisk (*) in its name. The asterisk must be replaced by one or more modifiers to create a legal feature name. For example, the global reduction pseudo-base feature **g*R** has modifiers `f`, `o`, `x`, `m`, `s`, and `p`. These represent the features **gfR**, **goR**, **gxR**, **gmR**, **gsR**, and **gpR**.

3.2.2 Communication

Interprocessor communication is a very important characteristic of SIMD architectures. It is the area of SIMD architecture which exhibits the most diversity. The primary determinant of an architecture's communication capabilities is the network used to connect the processors. A cartesian grid with one or two dimensions is most common, but complete graphs and other networks have also been proposed and used. The "labeling" and "communication" feature categories define the basic interprocessor communication network.

These categories make no pretense of describing all the possible communication networks that could be used in SIMD architectures. There is an infinite number of such networks, and a tremendous body of literature describing them. (Two good surveys are [WuFeng84, Siegel85].) This taxonomy only attempts to describe interconnection networks which have been used or proposed as part of a SIMD architecture. Obviously, more features will be added to the "labeling" and "communication" categories as new SIMD architectures use additional interconnection networks.

In some architectures, each PE is able to independently choose with which one of several (or many) other PEs it will communicate. This raises the possibility that some PE will be

Communication

Labeling (number, N)

1	1-D cartesian coordinate.
2	2-D cartesian coordinate.
...	Any fixed-dimension cartesian coordinate.
N	N-D cartesian coordinate.
p2	3-tuple: level number and 2-D cartesian coordinate.

Communication (C)

aC		Adjacent communication.
	d	Diagonal.
	w	Wrap.
	l	Local.
mC		Pyramid of 2-D grids, adjacent communication.
	c	Count.
cC		Cube Connected Cycles communication.
pC		Preselected set of permutations.
gC		Global communication.

Collision Resolution, Write (W)

sW	Select.
lW	Logically combine (and, or, xor).
aW	Add, or take maximum or minimum.
mW	Multiply.

Collision Resolution, Fetch (F)

sF	Select.
rF	Replicate.

Piped Communication (P)

P		Pipe.
	u	Unrestricted pipe.
	c	Copy pipe.

Cut-Through Communication (T)

T		Cut-through communication.
	r	Restrict the switch.
	l	Program switch locally.
	c	Connect through switch.
	o	"Or" together multiple values.

Table 3.1: Communication Features of SIMD Architectures. Summary of the communication group of features described in section 3.2.2, shown by category.

Local Addressing		
Local Addressing (L)		
	lL	Limited local addressing.
	uL	Unlimited local addressing.
	c	Local addressing during communication.
Reducing and Scanning		
	f	First.
	o	Or, and.
	x	Xor.
	m	Max, Min, f.
	s	Sum (add), m, x, o.
	p	Product (multiply), s.
Reduce (R)		
	d*R	Reduce along one dimension.
	m*R	Reduce along multiple dimensions.
	g*R	Global reduction.
Scan (S)		
	d*S	Scan along a dimension.
	g*S	Global scan.
Parallel I/O		
Input (I)		
	I	Parallel input.
Output (O)		
	dO	Parallel output, write-only (display).
	O	Parallel ouput.
PE to Host I/O		
Get (G)		
	rG	Restricted get.
	G	Get.

Table 3.2: Other Features of SIMD Architectures. Summary of non-communication features described in sections 3.2.3–3.2.6, shown by group and category.

selected as the communication partner of more than one other PE. How such communication collisions are resolved is determined by the "write" and "fetch" feature categories.

"Piped communication" extends ordinary grid communication by allowing PEs to communicate with PEs which are a fixed distance and direction away, but the distance is greater than one.

The feature categories described so far control the communication pattern either globally in the host, or locally in each originating PE. The final communication category, "cut-through communication", provides distributed control of communication. Each PE may initiate, forward, or intercept data. This has the interesting consequence that neither the host nor the originating PE necessarily knows with which other PE an originating PE is communicating.

3.2.2.1 Labeling (number, N)

The n PEs of every SIMD architecture are numbered from 0 to $n-1$. PE numbers are used by operations described later in this taxonomy. An architecture may define a supplementary labeling of PEs, used to identify individual PEs in certain operations provided by the architecture. There must be a mapping from PE numbers to PE labels which is one-to-one and onto. However, the definition of this mapping is an implementation detail not defined by the architecture. The following labelings are defined.

1 1-D cartesian coordinate.

2 2-D cartesian coordinate.

... A cartesian coordinate with any fixed number of dimensions.

N N-D cartesian coordinate. An architecture with this labeling specifies no fixed number of dimensions. A program may specify the number of dimensions to be used, up to some implementation-defined maximum.

p2 A 3-tuple consisting of a 2-D cartesian coordinate and a level number. This is used with the pyramidal multi-grid "mC" communication feature described in the next section. The PEs are arranged in 2-D square cartesian grids of fixed sizes. The grids are numbered $0, 1, 2, \ldots$. The grid at level l has 2^l PEs along each side. The label's level number specifies a grid, and the 2-D coordinate specifies a PE within that grid.

In the absence of the "gC" global communication feature and of features which require a labeling, these labelings have no significant effect and are all equivalent. Their real importance is in the way they influence certain other features whose behavior depends on the number of dimensions. However, "N" by definition subsumes any fixed-dimension cartesian labeling ("1", "2", ...) whenever the remaining features are kept constant. Any fixed-dimension cartesian labeling subsumes any smaller-dimensioned labeling when the remaining features are constant and do not include certain features. The excluded features are: all piped and cut-through communication features, and the dimensioned (non-global) reduce and scan features.

3.2.2.2 Communication (C)

The following characteristics are shared by all features in this category. When a communication operation is performed, all enabled PEs initiate communication with a remote PE. (The identity of the remote PE is specified differently in each feature.) The communication operation may send data to the remote PE, or fetch data from the remote PE. In both cases, the operation is unaffected by the enable status of the remote PE.

aC Adjacent communication. This feature requires a cartesian coordinate labeling. Each enabled PE may communicate with PEs whose labels differ from its own by exactly one in exactly one dimension. It may communicate with only one of them at a time, and the dimension and direction (+1 or −1) of the other PE are specified globally. It has three modifiers.

- **d** Diagonal. Each enabled PE may communicate with PEs whose labels differ from its own by exactly one in one or more dimensions. "d" also modifies piped and cut-through communication features.
- **w** Wrap. Label differences in each dimension may be calculated modulo the length (in PEs) of that dimension, so that the ends of each dimension wrap around. "w" also modifies piped and cut-through communication features. The choice of whether to wrap or not is made globally for each operation.
- **l** Local. The dimension(s) and direction(s) (+1 or −1) of communication are chosen locally (i.e., independently in each enabled PE). "l" does not modify any other features.

These modifiers may be combined freely, yielding the following features: aC, adC, awC, alC, adwC, adlC, awlC, adwlC. Figure 3.1 illustrates the interprocessor communication networks used for adjacent communication, each labeled with its feature name. Note that "l" affects the way the network is used, but not the topology of the interconnection network itself.

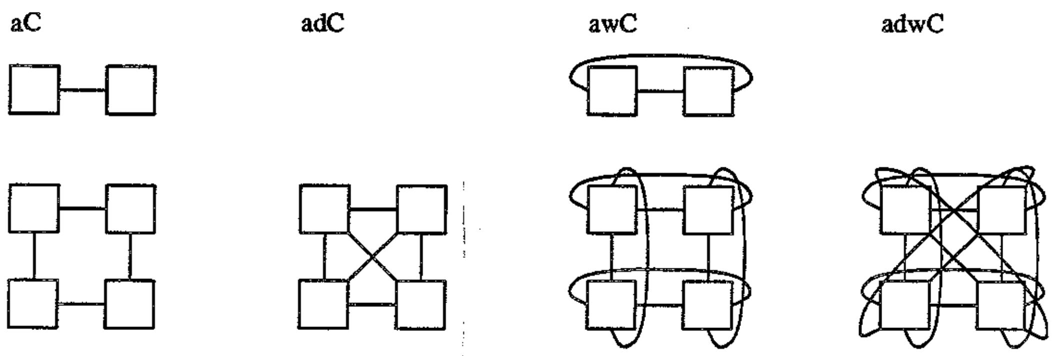

Figure 3.1: 1-D and 2-D Adjacent Communication Networks

For any fixed number of dimensions, architectures with and without the "l" modifier are equivalent. They are not equivalent when the "N" cartesian labeling is used. For any fixed number of dimensions, architectures with and without the "d" modifier are equivalent when they contain no piped or cut-through communication features. Architectures with and without the "w" modifier are equivalent if they do not contain any piped or cut-through communication feature. Note that "w" makes wrap-around communication available, but does not require its use. When wrap-around communication is available, each communication operation specifies whether to use it.

mC Multiple 2-D grids with pyramidal interconnections. This feature requires the "p2" pyramidal labeling. PEs are connected between levels so that each PE not on the last level is connected to 4 child PEs on the next level that form a 2 by 2 sub-grid. Each enabled PE may communicate with its child and parent PEs, and with its adjacent PEs on the same level (as defined by the "aC" feature for a 2-D cartesian grid). The choice of which of these PEs to communicate with is normally made globally.

The "d", "w", and "l" modifiers applied to "aC" are not relevant here. Because the number of neighbor PEs with which communication is possible is fixed, simulation of these modifiers is constant-bounded. So the description just given without these modifiers is equivalent to a description which includes them.

"mC" has one modifier.

c Count. Count the number of enabled PEs on the largest layer, returning the result to the host. This operation is included here because it is only applicable to architectures with this pyramidal topology.

The following features are legal: mC, mcC.

cC Cube Connected Cycles communication. Each enabled PE may communicate with PEs which are adjacent to it in the Cube Connected Cycles interconnection network as described in [PreparataV81]. The choice of which of these possible PEs to communicate with is made locally. (This is equivalent to making the choice globally, since the number of possibilities is fixed at three.)

pC Preselected set of permutations. A globally selected permutation specifies which pairs of PEs may communicate, and which PE of each pair initiates communication (if it is enabled). The maximum number of permutations which may be used by a single program is implementation dependent.

gC Global communication. Each enabled PE may communicate with any PE it chooses.

Under certain conditions, this feature subsumes each of the other features in the communication (C) category. First, any labeling required by the other feature must be present. Second, the "cP" piped communication feature must not be present, and no cut-through communication feature may be present. (The cut-through communication features provide a distributed routing mechanism in which the originating PE does not necessarily know the identity of the PE it communicates with.)

If a cartesian labeling is present with "gC", then the "aC" feature is implicitly present also. The "aC" modifiers "d" and "w" may be prepended to "gC" in this case. This does not change the architecture, since "adwC" is subsumed by "gC" when a labeling is present. It is significant because it allows the implicit "aC" to be modified by a piped or cut-through communication feature. It also allows the "aC" modifiers "d" and "w" to modify that piped or cut-through communication feature. The MP-1 (section 3.3.19) is an example of this. Piped and cut-through communication features require a variant of "aC", are modified by the "d" and "w" modifiers of "aC", and do not affect "gC".

3.2.2.3 Collision Resolution, Write (W)

When more than one PE attempts to send (write) data to a single PE, there is a write collision. If none of the features in this category are present, the result of a write collision is undefined. Collision resolution does not apply to piped communication (P) operations, which do not result in collisions. Collision resolution also does not apply to cut-through communication (T) operations, which handle collisions differently.

Write collisions can occur with any of the communication (C) features except "pC" (preselected permutation communication). (All "pC" communication is by permutation, so collisions are impossible.) Collisions can occur with "gC" (global communication), with "aC" (adjacent communication) modified by "l" (local direction), and with "mC" (pyramidal communication) or "cC" (CCC communication) when local communication is in use.

Collisions can also occur in "mC" whenever more than one child of a single parent PE attempts to communicate with that parent PE.

sW Select. One of the values is selected arbitrarily and used, and the other values are ignored. This is the same as serializing the write operations.

lW Logically combine. The inputs are combined with one of the bitwise logical operations "and", "or", or "xor". The behavior of "sW" may also be used. The choice of operator and behavior is made globally.

aW Add, or take maximum or minimum. The inputs are combined with one of the arithmetic operations "add", "max", or "min". The behavior of "lW" may also be used. The choice of operator and behavior is made globally.

mW Multiply. The inputs are combined with multiplication. The behavior of "aW" may also be used. The choice of behavior is made globally.

Any architecture with a fixed limit on the number of colliding requests can do constant-bounded simulation of all these features. This includes all the fixed-dimension cartesian coordinate labeled machines (not "N") which have variations of "aC" communication but do not also have "gC" communication. It also includes architectures with "cC" and "mC" communication.

3.2.2.4 Collision Resolution, Fetch (F)

When more than one PE attempts to fetch (read) data from a single PE, there is a fetch collision. If none of the features in this category is present, the result of such an operation is undefined. Collision resolution does not apply to piped communication (P) operations, which do not result in collisions. Collision resolution also does not apply to cut-through communication (T) operations, which handle collisions differently.

Fetch collisions can occur under the same circumstances as write collisions.

sF Select. One of the PEs requesting data will receive it. The others will not receive any data, leaving the location where it was to be stored unmodified.

rF Replicate. All the requesting PEs receive a copy of the data. This is the same as serializing the requests. The behavior of "sF" may also be used. The choice of behavior is made globally.

"sF" is subsumed by "sW", as follows. Each PE simulating "sF" sends to the PE it is fetching from some data which identifies the sending PE (e.g., PE number or relative position). The "sW" feature ensures that each PE receives at most one such message. Each receiving PE simply sends the fetched data back to the PE identified by the request it received.

The same architectures which can do constant-bounded simulation of write collision resolution features can also do constant-bounded simulation of all fetch collision resolution features.

3.2.2.5 Piped Communication (P)

Piped communication is simply extended adjacent communication. Communication is with a PE some globally-specified number of PEs away, where that distance need not always be one as with adjacent communication. The "d" (diagonal) and "w" (wrap) modifiers of "aC" (adjacent communication) modify piped communication as well. The "l" (local direction) modifier of "aC" does not apply to piped communication; the choice of direction for piped communication is always made globally.

Piped communication only has meaning on architectures with adjacent communication (variations of the "aC" feature). Piped communication may not be used simultaneously with cut-through communication. An architecture may have features from both categories (though none that I know of does), but these features do not affect each other. Collision resolution does not apply to piped communication, because no collisions can occur. Local addressing during communication ("*cL") does apply to piped communication.

P Pipe. The restriction is imposed that there may be no enabled PE in the straight path between any two communicating PEs. "P" has two modifiers.

- **u** Unrestricted pipe. There is no restriction on which PEs may be enabled.
- **c** Copy pipe. During restricted pipe communication, copy the communicated data into each disabled PE in the straight path between each pair of communicating PEs.

The "u" and "c" modifiers may both be present, but they do not modify each other.

The legal features are: P, uP, cP, and ucP. "ucP" provides all three kinds of communication described in this category.

Global communication ("gC") subsumes "P" and "uP", but not "cP".

3.2.2.6 Cut-Through Communication (T)

Cut-through communication is an extension to adjacent communication. Also known as connection autonomy, cut-through communication places a crossbar switch at each PE's connection to the communication network. This switch can be globally or locally programmed, but the default is global. The switch can be the PE's connection to the network (in which case the switch is connected to the network all the time), or it can be used to temporarily replace the PE in the network at the PE's discretion. The latter case is the default: the PE may choose locally to disconnect itself from the network and connect its switch to the network in its place.

These characteristics allow the creation of multiple logical "wires", each connecting some number of PEs in an extremely flexible pattern. If any PE places data on such a wire, all the other PEs connected to the wire receive that data. If more than one PE sends data to the same wire, the result is undefined unless "o" ("or" combine, described later in this section) is specified.

Cut-through communication allows multi-cast communication, where the data sent by a single PE may be received by multiple remote PEs. For this reason, cut-through communication supports sending data, but not fetching data. That is, when each enabled PE initiates communication with some set of remote PEs, data is always sent from each enabled PE to its remote PEs. The enabled PEs never fetch data from remote PEs.

T Cut-through communication. Place a globally-programmable crossbar switch at each PE's connection to the interconnection network.

r Restrict the switch. Replace the crossbar with a rotary "straight-across" switch. This switch always connects one pair of network wires that go in opposite directions.

l Local switch programming. Allow each PE to program its own switch.

c Connect through switch. Expand the switch so the PE can connect to the network through the switch, instead of having to remove the switch from the network to connect itself to the switch. If the switch is a crossbar, simply add another port to it for the PE's connection. This connection is programmed with the rest of the switch. If the switch is the restricted rotary switch, add a connection from the PE to the center of the switch's straight-across wire. Whether this connection is made or broken is controlled in the same way as the switch's orientation, whether that is global or local.

o "Or" together multiple values. If more than one PE sends data to a single wire (set of connected PEs), the data is bit-wise "or"d together instead of being undefined.

The modifiers may be combined freely, so the legal features are: T, rT, lT, cT, oT, rlT, rcT, roT, lcT, loT, coT, rlcT, rloT, rcoT, lcoT, rlcoT.

Figure 3.2 shows some example communication patterns using cut-through communication. Each pattern is labeled with the feature it illustrates. The small rectangle on one side of PEs in some patterns indicates that the PE itself is connected to that communication port. Note that "o" affects the behavior of the network, but not the topology, so it is not visible in the figure.

Cut-through communication requires adjacent communication ("a*C"). The "d"(diagonal) and "w" (wrap) modifiers of "aC" also apply to cut-through communication. "d" provides additional ports on the switch, which go to diagonally adjacent PEs. "w" allows wrap-around wires between switches at opposite edges of the PE array. The "l" (local direction) modifier of "aC" does not apply to cut-through communication. The direction of communication is specified by the configuration of the switches.

Cut-through communication may not be used simultaneously with piped communication. An architecture may have features from both categories (though none that I know of does), but these features do not affect each other. Collision resolution features (W, F) do not apply to cut-through communication; collision handling for cut-through communication is described earlier in this section. Local addressing during communication ("*cL", section 3.2.3) does not apply to cut-through addressing; all addressing during cut-through communication is global.

This taxonomy could easily be extended to allow cut-through communication with the "mC" (pyramidal communication) and "cC" (CCC communication) communication features. But no such architecture has been proposed, so that extension has not been done yet.

Global communication ("gC") does not subsume cut-through communication. Cut-through communication does not subsume piped communication or global communication.

Locally programmed ("l") cut-through communication raises an interesting implementation difficulty. The host does not necessarily know the length of the longest wire carrying data in a communication operation, because that depends on the locally programmed switch configurations. Even in the absence of "l", the host may still not know the length of the longest data wire if the "c" modifier is also not present, because the wire length may depend on which switches are connected to the network by their PEs. Not knowing the length of the longest data wire makes it difficult to choose the right length of time to wait for the data to propagate on the wire. The implementation of YUPPIE (Yorktown Ultra Parallel Polymorphic Image Engine, section 3.3.13) handles this by requiring the program to specify a bound on the length of the longest wire in each communication operation.

3.2.3 Local Addressing

Local memory within each PE is most commonly accessed using the same globally-specified address in all PEs. Allowing PEs to access their local memory using different locally-

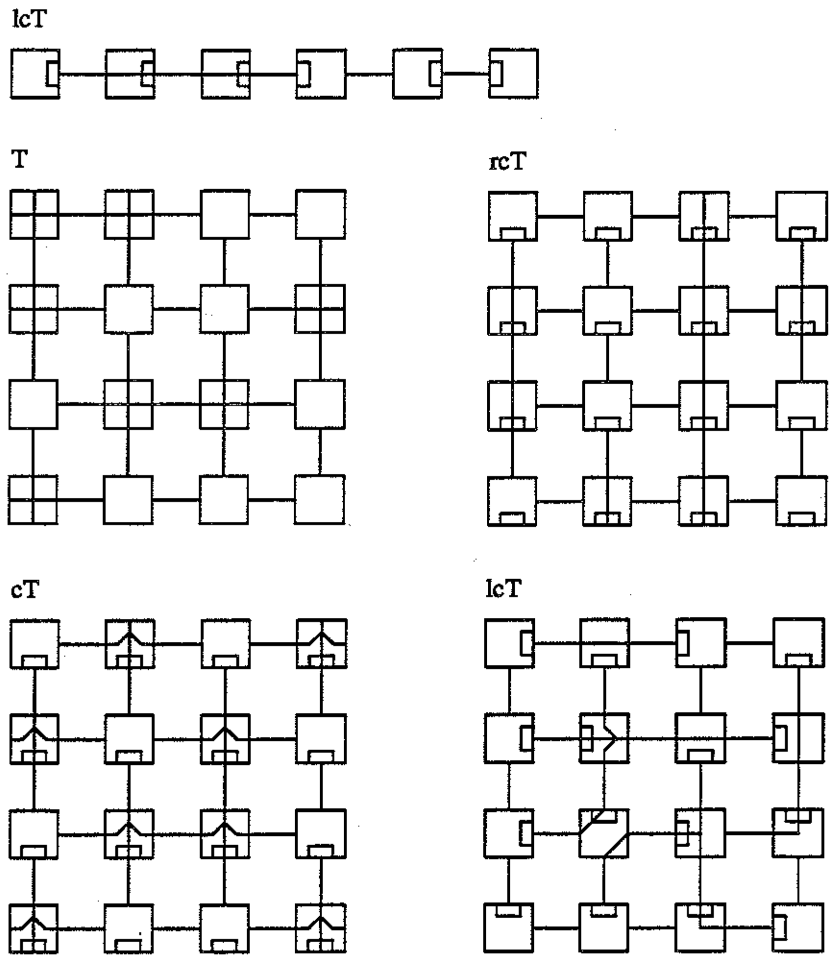

Figure 3.2: Examples of Cut-Through Communication

computed addresses adds significantly to an architecture's flexibility and power. Middleton and Tomboulian discuss the costs and benefits of local addressing [MiddletonT89].

3.2.3.1 Local Addressing (L)

lL Limited local addressing. Each PE can locally address some subset of its memory.

uL Unlimited local addressing. Each PE can locally address its entire memory. This obviously subsumes "lL".

c Local addressing during communication. Each PE initiating a communication operation may specify that the remote PE with which it is communicating use local addressing to load or store the data, as appropriate. The initiating PE specifies the address which the remote PE is to use, and sends this address to the remote PE as part of the communication operation. The form of local addressing provided ("lL" or "uL") is the same as that available in the local PE.

This modifier is only meaningful in an architecture with a communication (C) feature. The collision resolution (W, F) features apply to local addressing during communication, except that two requests are not considered to collide unless they specify both the same remote PE and the same address. Local addressing during communication is allowed with piped communication (P). However, local addressing during communication does not apply to cut-through communication (T).

"c" modifies both "lL" and "uL", so the legal features are: lL, uL, lcL, ucL.

CAM (Content Addressable Memory) is sometimes associated with SIMD architectures. However, there is an important distinction between a SIMD architecture used to implement a CAM, and a SIMD architecture with CAM words as its local memory. The former is common in the literature, and uses only global addressing in each PE. The latter has never been proposed, to my knowledge.

3.2.4 Reduce and Scan

Reduce and scan are similar in that they both apply some combining operator to a value from each enabled PE in some set of PEs. Reduce is concerned only with the final result, while scan also computes and saves each intermediate result. Scan is also known as the data independent prefix operation.

Computing reduce and scan with fixed fan-in operators clearly takes $\log n$ parallel steps for n PEs. However, both operations are provided as single operations on commercially available SIMD computers, as shown in section 3.3. Reduce and scan are actually at least as efficiently implemented as the global communication operations which are also provided as single operations on the same and additional computers. Blelloch elaborates on this point [Blelloch87].

Reduce and scan share the following modifiers, which specify sets of combining operators.

f	First. Always use the first (leftmost) operand as the result. For scan, this is also called copy.
o	Or, and. Bitwise inclusive "or", or bitwise "and".
x	Xor. Bitwise exclusive "or", or use one of the "o" operations.
m	Max, Min. Maximum, minimum, or use the "f" operation.
s	Sum. Add, or use one of the "m", "x", or "o" operations.
p	Product. Multiply, or use one of the "s" operations.

3.2.4.1 Reduce (R)

d*R Reduce along one dimension. Place in each enabled PE the result of combining the values of all enabled PEs whose labels differ from that PE's label in only one dimension. This requires a cartesian labeling.

m*R Reduce along multiple dimensions. Place in each enabled PE the result of combining the values of all enabled PEs whose labels differ from that PE's label only in the specified dimensions. This requires a cartesian labeling.

g*R Global reduction. Return to the host the result of combining the values of all enabled PEs.

These are all pseudo-features, meaning that they are not real features without a modifier to specify the combining operation. The "d", "m", and "g" prefixes may be combined. They do not modify each other, so combining them simply makes the different features available within the same architecture. Since "m" subsumes "d", there is no reason to write them both. Therefore, the legal features are: dfR, doR, dxR, dmR, dsR, dpR, mfR, moR, mxR, mmR, msR, mpR, gfR, goR, gxR, gmR, gsR, gpR, dgfR, dgoR, dgxR, dgmR, dgsR, dgpR, mgfR, mgoR, mgxR, mgmR, mgsR, mgpR.

The features "goR" and "gmR" are equivalent. The "m*R" features subsume the corresponding "d*R" features, by definition. Each "d*R" feature subsumes the corresponding "g*R" feature for any fixed number of dimensions (e.g., cartesian labelings "1", "2", ..., but not "N").

3.2.4.2 Scan (S)

d*S Scan along a dimension. Place in each enabled PE the result of combining its value with the values of all enabled PEs whose labels differ from that PE's label in only one dimension and have a lower number in that dimension. This requires a cartesian labeling.

g*S Global scan. Place in each enabled PE the result of combining its value with the values of all enabled PEs with a lower PE number than its own.

These are all pseudo-features, meaning that they are not real features without a modifier to specify the combining operation. The "d" and "g" prefixes may be combined. They do not modify each other, so combining them simply makes the different features available within the same architecture. Therefore, the legal features are: dfS, doS, dxS, dmS, dsS, dpS, gfS, goS, gxS, gmS, gsS, gpS, dgfS, dgoS, dgxS, dgmS, dgsS, dgpS.

The dimensioned copy scan ("dfS") subsumes piped copy communication ("ucP").

3.2.5 Parallel I/O

A parallel I/O operation transfers in a single operation a value for each PE between the PE and an external data source or sink. The transfer is performed in every PE, whether the PE is enabled or not.

3.2.5.1 Input (I)

I Parallel input. Read a value into each PE from a data source.

Typical data sources include disk files and video devices.

3.2.5.2 Output (O)

dO Parallel output to a write-only device. Write a value from each PE to a write-only data sink.

O Parallel ouput. Write a value from each PE to a read/write mass storage device.

"dO" is named for "display", since some video displays are write-only devices. Disk files are the most common read/write mass storage devices.

3.2.6 PE to Host I/O

Architectures which do not provide parallel I/O usually provide serial I/O to and from the PEs through the host. Data can be sent from the host to any or all PEs in all SIMD architectures, by broadcasting the data as part of the instruction stream. The data can be preceded by instructions to ensure that exactly the right PEs are enabled and therefore able to record the data from the host. Getting data from PEs back to the host is a very common feature, but it is not universal.

3.2.6.1 Get (G)

rG Restricted get. Copy a value from some unspecified arbitrary PE to the host.

G Get. Copy a value from any specified PE to the host.

"G" implies full serial I/O to and from the PEs through the host.

The feature "goR" (global "or" reduction) subsumes "G" (get).

3.2.7 Naming a Classification

An architecture's classification is written by concatenating the names of its features. The feature names should normally be written in the same order used to describe them in the preceding sections, as a convenience to the reader.

3.3 The Taxonomy in Use

This section shows that the taxonomy just presented is useful and reasonably comprehensive, by using the taxonomy to classify a representative set of 20 SIMD architectures. The architectures and their classifications are summarized in table 3.3 and discussed in the remainder of this section.

These architecture classifications are presented as a tree in figures 3.3 and 3.4. The root node of the tree is the basic SIMD architecture with no optional features. The leaves are real computer systems; each leaf system implements the architecture containing the set of optional features named on the path from the root to that leaf. Each vertical level of the tree is labeled with the feature category it represents, and each box in a level names a particular feature in that category.

In both the table and tree formats, the *canonical* classification of each architecture is shown (as *canonical* is defined in the next paragraph). However, some architectures are listed again in parenthesis to point out architectures which it may not be obvious that they subsume. Note that the power of the architectures does not increase monotonically, as the orderings might suggest. Constant-bounded simulation (the "subsumes" relation) provides only a partial ordering.

The architecture classifications that follow are presented in a common format. The header is the name of the architecture, with a parenthesized explanation of the name if appropriate. This is followed by the architecture's classification. The classification is stated at least twice. The first statement is the *most natural* classification. The second is equivalent to the first, but is a *canonical* classification which names all features the architecture can provide with constant-bounded simulation. This canonical form is used in table 3.3. Any succeeding classification statements show important subsumed architectures which might otherwise be overlooked.

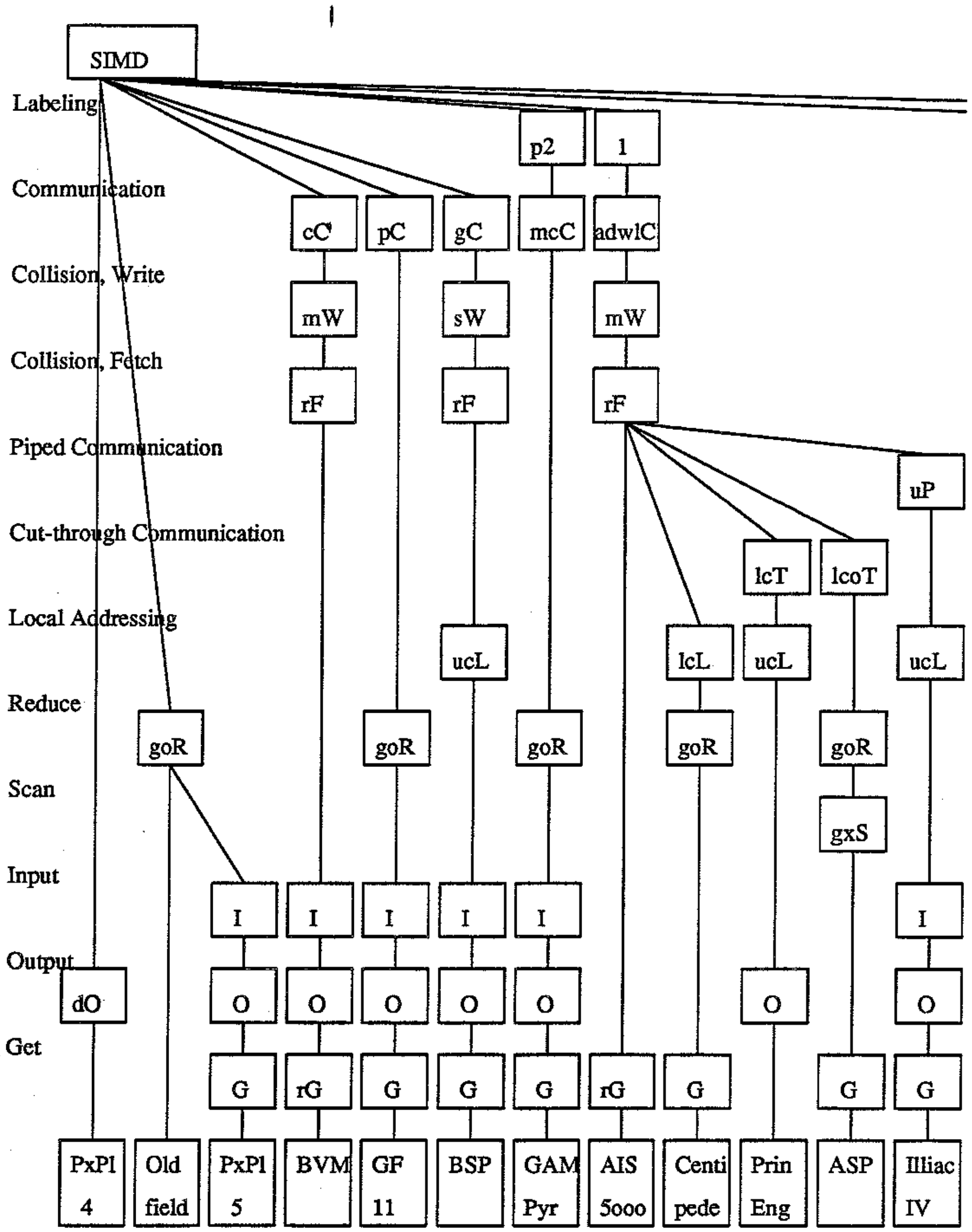

Figure 3.3: Tree of SIMD Architecture Classifications. The remainder of the tree is shown in the next figure.

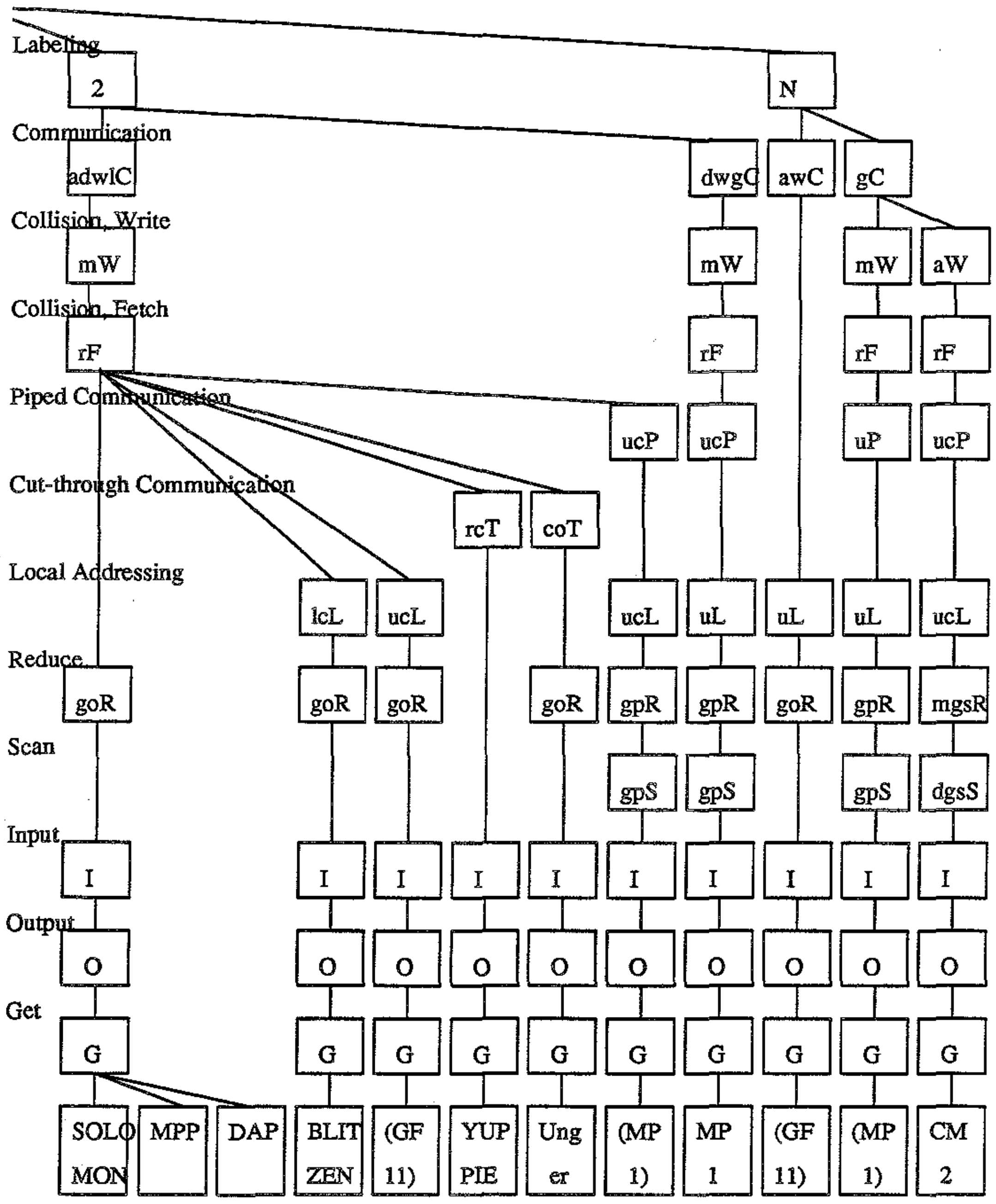

Figure 3.4: Tree of SIMD Architecture Classifications, continued from the previous figure.

Name	.N	C	.W	.F	..P	...T	..L	...R	...S	I	.O	.G
PxPl4											dO	
Oldfield								goR				
PxPl5								goR		I	O	G
AIS-5000	1	adwlC	mW	rF								rG
Centipede	1	adwlC	mW	rF			lcL	goR				G
Princeton Engine	1	adwlC	mW	rF		lcT	ucL				O	
ASP	1	adwlC	mW	rF		lcoT		goR	gxS			G
Illiac IV	1	adwlC	mW	rF	uP		ucL			I	O	G
SOLOMON	2	adwlC	mW	rF				goR		I	O	G
MPP	2	adwlC	mW	rF				goR		I	O	G
DAP	2	adwlC	mW	rF				goR		I	O	G
BLITZEN	2	adwlC	mW	rF			lcL	goR		I	O	G
YUPPIE	2	adwlC	mW	rF		rcT				I	O	G
Unger	2	adwlC	mW	rF		coT		goR		I	O	G
GAM Pyramid	p2	mcC	mW	rF				goR		I	O	G
BVM		cC	mW	rF						I	O	rG
GF11		pC					uL	goR		I	O	G
(GF11)	N	awC					uL	goR		I	O	G
(GF11)	2	adwlC	mW	rF			ucL	goR		I	O	G
BSP		gC	sW	rF			ucL			I	O	G
MP-1	2	dwgC	mW	rF	ucP		uL	gpR	gpS	I	O	G
(MP-1)	2	adwlC	mW	rF	ucP		ucL	gpR	gpS	I	O	G
(MP-1)	N	gC	mW	rF	uP		uL	gpR	gpS	I	O	G
CM-2	N	gC	aW	rF	ucP		ucL	mgsR	dgsS	I	O	G

Table 3.3: Classification of SIMD Architectures. Canonical classifications are shown. Classifications with the architecture name in parentheses show significant subsets which are subsumed by the canonical classification, but are not obvious from it.

The text following each architecture's classification mentions any distinctive features that influenced the design of the taxonomy. The lowest-level publicly documented programming interface that defines the architecture is identified. The organizational affiliation and implementation status are given, if known. References describing the architecture are given last.

3.3.1 PxPl4 (Pixel-Planes 4)

2 dO
dO

I believe Pxpl4 is the most architecturally minimal SIMD machine proposed, but also the most massively parallel machine ever built (256K PEs). Its one optional feature is "dO" (display output), which enables it to display PE values directly to a video display. PxPl4 is unique in not providing a way to get data from the PEs back to the host, and "dO" was created to describe these limited I/O capabilities. PxPl4's most powerful and unique feature is a fast hardware bilinear expression evaluation tree. Because this provides a large but constant savings in operations, it is not apparent in this taxonomy. The 2-D labeling is natural because it is supported by the hardware expression evaluation tree and reflects the video output format. However, the labeling is architecturally extraneous.

A hardware-specific C-callable library defines the architecture. Designed and built in the University of North Carolina at Chapel Hill's Department of Computer Science, a 256K PE system has been running since 1986. [FuchsPoult81, FuchsGHSA+85, EylesAuFGP87, GreerEyles88]

3.3.2 Oldfield (Oldfield et al.)

			goR		
			goR		G

This machine is very close to being the minimum SIMD architecture. It is a CAM (Content Addressable Memory) with enough logic in each word for the word to barely qualify as a SIMD PE. Constant-bounded simulation of arithmetic operations and an enable bit is laborious, but possible. Memory is very limited, since the architecture specifies only one word of memory and a few 1-bit registers. Such limited memory is typical of CAMs, even those that qualify as SIMD architectures.

The architecture is described in terms of very low-level operations, but no software interface is described in the references I have. The authors are at Cambridge and Syracuse Universities. They report that the architecture has been simulated on a Connection Machine. [OldfielWWB88]

3.3.3 PxPl5 (Pixel-Planes 5)

2	goR	I	O	G
	goR	I	O	G

The successor to PxPl4, PxPl5 also eschews communication and achieves the same degree of parallelism. However, PxPl5 is more mainstream in its I/O system. The hardware expression tree evaluates biquadratic expressions, providing a large but constant savings in operations. A hardware-specific C-callable library defines the architecture. A 256K PE PxPl5 system is under construction in the University of North Carolina at Chapel Hill's Department of Computer Science. [FuchsPEGG+89, UNC MSL88]

3.3.4 AIS-5000 (Applied Intelligent Systems, Inc.)

1	aC	rG
1	adwlC mW rF	rG

This architecture has no enable register, but can simulate one. A hardware-specific C-callable library defines the architecture. Applied Intelligent Systems, Inc. appears to be selling AIS-5000 systems consisting of 128 to 1024 PEs. [Wilson88]

3.3.5 Centipede

1	aC	lL	goR	rG
1	adwlC mW rF	lcL	goR	G

This is a successor to the AIS-5000, differing from the AIS-5000 primarily by adding limited local addressing and or-reduction. Its local addressing allows each PE to specify the low-order address bits, while the higher-order bits are specified globally. Centipede is a planned commercial product of Applied Intelligent Systems, Inc. [Wilson88]

3.3.6 Princeton Engine

1	aC	lcT	uL	O
1	adwlC mW rF	lcT	ucL	O

The locally programmed, connect through switch form of cut-through communication ("lcT") used by the Princeton Engine is not seen in any other machine to my knowledge. It is similar to that used by the ASP, though. Another apparently unique feature is having parallel output but not parallel input. Input is performed by shifting in one end of the 1-D array of PEs. But output is in parallel to a buffer to which video output devices have random access. This buffer is a read/write mass storage device, but it is unusual that the Princeton Engine cannot read from it in parallel. (But the output data could be sent back to the sequential input port.)

The architecture is defined by its microcode. The Princeton Engine was designed and built at the David Sarnoff Research Center. At least two systems, each with up to 2000PEs, were built in 1989. [ChinPasBTK88, Taylor88]

3.3.7 ASP (Associative String Processor)

1	aC	lcoT	goR	gxS	G
1	adwlC mW rF	lcoT	goR	gxS	G

The locally programmed, connect through switch, "or" combining form of cut-through communication ("lcoT") used by the ASP is similar to that used by the Princeton Engine, but not identical. Like the Oldfield architecture, the ASP is a SIMD CAM with only one word of memory per PE. The ASP's "disable every other enabled PE" feature is equivalent to "gxS" (global "xor" scan), and prompted the inclusion of this feature in the taxonomy.

A hardware-specific C-callable library defines the architecture. Chips have been designed at Brunel University and Aspex Microsystems Ltd, but apparently no system has been built yet. (The benchmarks referenced appear to be based on simulations.) [Lea86b, Lea88, KrikelisLe88b, KrikelisLe88a, Lea86a, KrikelisLe88c]

3.3.8 Illiac IV

1	awC	uP	uL	I	O	G
1	adwlC mW rF	uP	ucL	I	O	G
2	adwlC mW rF	uP	ucL	I	O	G

The Illiac IV is usually considered to have a 2-D grid topology. However, it also supports a 1-D ring with unrestricted piped communication ("uP"), which is more powerful. This 1-D architecture subsumes the 2-D architecture, simulating 2-D shifts with 1-D shifts of distance one or the length of a side of the 2-D grid. The Illiac IV prompted the "uP" feature in the taxonomy. This feature is also supported by the MasPar MP-1. (Though the CM-2 subsumes "uP" with its global communication, it does not support it directly.)

The Illiac IV's architecture is defined by the ASK assembly language. The Illiac IV is the best known early SIMD architecture. It was conceived by Daniel Slotnick, who had previously been involved in the SOLOMON (see section 3.3.9) design effort. Though installation at the University of Illinois was initially planned for 1970, campus turmoil caused a change in location. The Illiac IV, with 64 PEs, was delivered to the NASA/Ames Research Center in 1972. After a period of software development and hardware debugging, it officially became operational in 1975. [Hord82]

3.3.9 SOLOMON (Simultaneous Operation Linked Ordinal MOdular Network)

2 adwlC		goR	I O G
2 adwlC mW rF		goR	I O G

SOLOMON was the first SIMD computer ever built. It introduced the concept of the enable bit. I think it is a remarkable tribute to SOLOMON's designers that the MPP and DAP still use an identical architecture today.

SOLOMON's architecture is defined by an assembly language. Westinghouse Electric Corporation designed and built SOLOMON. A 9 PE machine was operational in 1963. A 1024 PE machine was planned, but to my knowledge was not built. [SlotnickBM62, BallBolJMS62, Schaefer90]

3.3.10 MPP (Massively Parallel Processor)

2 awC		goR	I O G
2 adwlC mW rF		goR	I O G

The architecture is defined by the MCL assembly language and the Pearl microcode assembly language. A 16K PE MPP was developed by Goodyear Aerospace Corporation for NASA Goddard Space Flight Center and delivered in 1983 and recently decommissioned. [Potter85, NASA/Godda85b, NASA/Godda85a]

3.3.11 DAP (Distributed Array Processor)

2 awC		goR	I O G
2 adwlC mW rF		goR	I O G

There is a hint in [Active Mem88, p. 14] that the DAP may also support unrestricted piped communication ("uP"). The DAP's architecture is defined by APAL (Array Processor Assembler Language). Unfortunately, the appendix on APAL in [Active Mem88], which would be expected to clear up the question of "uP" support, omits all mention of communication. I do not have access to a copy of [Active Mem87a], which should provide an authoritative answer.

AMT (Active Memory Technology, Inc.) sells 1K and 4K PE DAPs commercially. Previous versions of the DAP were developed and built by ICL (International Computers Ltd.). [ParkinsoHM88, Active Mem88, Active Mem87a] [Iliffe82, chapter 12]

3.3.12 BLITZEN

2 adC	lL	goR	I O G
2 adwlC mW rF	lcL	goR	I O G

The BLITZEN architecture was designed as a successor to the MPP. The only change visible at the level of this taxonomy is the addition of local addressing. Whether this addressing is limited or unlimited depends on the system built from BLITZEN chips. Each PE can locally address its on-chip memory. However, the architecture provides for additional off-chip memory which is not locally addressable. That is why it is classified as "lL" (limited local addressing) here. A system which did not provide off-chip memory could be considered to support "uL" (unlimited local addressing) instead.

BLITZEN's architecture is defined by the BLITZ assembly language [RosenbergD88]. The BLITZEN project involves cooperation between researchers at MCNC, Duke, NCSU, and UNC. The architecture has been simulated [RosenberBH88], and chips have been fabricated at MCNC. A prototype system with 128 PEs on a single chip is being built at UNC. [BlevinsDHR90, DavisReif88, BlevinsDaR87]

3.3.13 YUPPIE (Yorktown Ultra Parallel Polymorphic Image Engine)

2 aC	rcT		I O G
2 adwlC mW rF	rcT		I O G

The papers that present the YUPPIE architecture have a dual purpose: to propose connection autonomy as a new feature for SIMD architectures, and to present YUPPIE as a demonstration of that feature. Connection autonomy as they describe it is equivalent to "lcT" (locally programmed, connect through switch, cut-through communication). However, YUPPIE only implements "rcT" (restricted, connect through switch, cut-through communication), which is dramatically simpler than "lcT". YUPPIE's usefulness as a demonstration of connection autonomy is therefore highly questionable. Both "lcT" and "rcT" are new features for 2-D cartesian grids, although Unger, ASP, and the Princeton Engine implemented other forms of cut-through communication independently.

The YUPPIE architecture is defined by an assembly language. A chip has been fabricated, but no system has been built, that I know of. This work was done at IBM's T. J. Watson Research Center. [MarescaLi88, MarescaLi89]

3.3.14 Unger (S. H. Unger)

2 adwlC	coT	goR	I O G
2 adwlC mW rF	coT	goR	I O G

S. H. Unger first proposed the concept of a SIMD computer in 1958 [Iliffe82, pp. 170 & 241]. The architecture he proposed did not include an enable bit, but is capable of simulating one with a constant number of operations. He did not call it a SIMD architecture, since that term was not coined by Michael Flynn until 1966 [Flynn66]. Besides inventing the idea of SIMD architectures, Unger proposed adjacent communication on a 2-D cartesian grid. He also proposed a form of non-local communication which is equivalent to the "coT" (connect through switch, "or" combining cut-through communication) feature. This feature has not been used since, to my knowledge.

The architecture is defined at the level of assembly language. Unger did this work at Bell Telephone Labs, Inc. The architecture was not implemented. [Unger58]

3.3.15 GAM Pyramid (George Mason University, Adder pyramid, MPP circuits, Pyramidal topology)

p2 mcC		goR	I O G
p2 mcC mW rF		goR	I O G

The GAM Pyramid prompted the "mC" (pyramidal communication) feature. The adder pyramid prompted the "c" (count) modifier of "mC", and is separate from the pyramid of PEs. The GAM Pyramid architecture is defined by a microcode assembler. The GAM II Pyramid (same architecture, second hardware implementation) contains 1365 PEs in six layers and became operational in 1988 in the labs of George Mason University. [Abuhamdeh88, SchaefeHBV87]

3.3.16 BVM (Boolean Vector Machine)

cC		goR	I O G
cC mW rF		goR	I O G

The BVM is the only SIMD machine I am aware of to use a Cube Connected Cycles interconnection network. This network occupies an interesting position, intermediate in power between the 1-D and 2-D grid architectures and the global communication architectures. For example, reduce and scan are not single operations, but they can be implemented with $O(\log(n))$ operations instead of the $O(\sqrt{n})$ required on 2-D grids without piped communication.

The BVM's architecture is defined by BVL-0, a C-like language with embedded assembly-level operations for the BVM [TombouliMW85, Tuck85, Tuck87]. The BVM was designed by Robert Wagner in the Duke University Department of Computer Science. A chip was fabricated and a small prototype system was built in 1986. [Wagner83, Wagner81]

3.3.17 GF11 (11-Gflop target performance)

pC	uL	goR		I	O	G
pC	uL	goR		I	O	G
N awC	uL	goR		I	O	G
2 adwlC mW rF	ucL	goR		I	O	G

The GF11's pre-programmable circuit switch is unique among SIMD machines. It allows a global choice from among 1K permutations selected and programmed into the switch during program startup. The GF11 architecture is defined by microcode embedded in Pascal. A GF11 built to accommodate a maximum of 576 PEs has been running with at least 400 PEs at IBM T. J. Watson Research Center since 1989 or before. [BeetemDenW85, BeetemDenW86, Wirbel89]

3.3.18 BSP (Burroughs Scientific Processor)

gC sW rF	ucL			I	O	G
gC sW rF	ucL			I	O	G

As already discussed in section 3.1, the BSP has a global memory connected to the PEs by a communication network. This is unique, to my knowledge. But it is equivalent to the architecture's classification, in which PEs have local memory and communicate with each other using the "gC" (global communication) feature. Collision resolution is by serialization only. This is not true of any other architectures I am aware of, but is not surprising in light of the underlying model of shared global memory.

I suspect that the BSP architecture is defined by an assembly language, but have not been able to verify this. The BSP was a commercial product of Burroughs Corporation, first sold in 1979, and contained 16 PEs. The BSP project grew out of Burroughs' involvement in the development of the Illiac IV. [HwangBrigg84, pp. 326-327 & 410-422] [Liebrick79]

3.3.19 MP-1 (MasPar Computer Corp.)

2 dwgC mW rF ucP	uL	gpR	gpS	I	O	G
2 dwgC mW rF ucP	uL	gpR	gpS	I	O	G
2 adwlC mW rF ucP	ucL	gpR	gpS	I	O	G
N gC mW rF uP	uL	gpR	gpS	I	O	G

Direct support for piped copy communication ("cP") appears to be unique, though it is subsumed by dimensioned copy scan ("dfS") in the CM-2. The combination of global communication with a 2-D grid ("2gC") is also both unique and yet subsumed by the CM-2's N-D grid ("NgC"). The "mW" (multiply to combine collisions on write) feature is a minor addition not found in other architectures with global communication. Of course, the 2-D labeling applies primarily to piped communication and dimensioned reduce and scan. So as the classifications above show, the MP-1 architecture subsumes some important "NgC..." architectures.

The MP-1 architecture is defined by MPL and its standard library. That library is still evolving, and the features "ucL" (unlimited local addressing during communication), "dpR" (dimensioned reduction with product) and "dpS" (dimensioned scan with product) might reasonably be added. Such architectural evolution through library enhancement has a precedent in the evolution of Paris, which added "msR" (multi-dimensional reduction with sum) and "dsS" (multi-dimensional scan with sum) in the upgrade from version 4 to version 5. The MP-1 is a commercial product of MasPar Computer Corp. and contains up to 16K PEs. [MasPar Com90b, MasPar Com90a]

3.3.20 CM-2 (Connection Machine, model 2)

N	gC	aW	rF	ucP	ucL	mgsR	dgs	I	O	G
N	gC	aW	rF	ucP	ucL	mgsR	dgs	I	O	G

The Connection Machine introduced, or at least popularized, several important architectural features. One is "N" (arbitrary dimensional cartesian labeling), which reflects the underlying hypercube used by the router. The combination of "gC" (global communication) and "aW" (add to combine collisions on write) is a more important contribution. Of its innovations, the provision of "mgsR" (multi-dimensioned or global reduction with sum) and "dgsS" (multi-dimensioned or global scan with sum) may have the largest influence on SIMD algorithms.

The Connection Machine was first embodied in the CM-1, which was replaced by the CM-2 in 1987. Although there are significant hardware differences in these models, their architectures as defined by Paris are similar or identical. Paris (Parallel Instruction Set) is a hardware-specific C-callable library. Different versions of Paris have defined slightly different architectures. The current version of Paris is 5.2, which runs only on the CM-2 and is classified here. Version 4.3 runs on both the CM-1 and CM-2. Thinking Machines Corp. sells CM-2s with up to 64K PEs. [Hillis85, Thinking M87b, Thinking M87a, Thinking M89]

3.4 Previous SIMD Taxonomies

No existing SIMD taxonomy I am aware of is a suitable base for the design of optimally portable SIMD languages. Of course, none was developed specifically for this purpose. Any comprehensive SIMD taxonomy which used constant-bounded simulation as its criterion for distinguishing architectural features would be suitable as a tool for optimally portable SIMD language design, but no existing taxonomy uses this criterion. This section gives a very brief survey of existing SIMD taxonomies.

Flynn [Flynn66] distinguishes and names the class of SIMD architectures. Feng [Feng77] and Handler [Handler77] propose some quantitative measures of parallelism, but do not further classify SIMD architectures. Snyder [Snyder88] extends Flynn's taxonomy to distinguish architectures with local addressing from those with only global addressing.

Rice et al. [RiceSeidmW88] propose a high-level classification of SIMD systems which deals primarily with the connections between the host, sequencer(s), PEs, and I/O subsystem. This classification does not deal with PE features such as local addressing, or with communication beyond identifying an interconnection graph.

Hwang and Briggs [HwangBrigg84, chapters 5-6] cover interconnection networks and associative memory well, and describe some particular SIMD architectures in detail. In the taxonomy section of their work, they do not recognize the presence or absence of local addressing in PEs as a significant architectural characteristic. There are other features they do not discuss at all. They distinguish between bit-serial and bit-parallel PEs, which are architecturally equivalent by the definition of chapter 2.

Siegel [Siegel85] develops a notation for describing SIMD architectures as part of his treatment of interconnection networks. The notation is used to describe the assumptions made in the remainder of that work on interconnection networks. It is not developed to distinguish other aspects of SIMD architectures.

Gerritsen [Gerritsen83] and Fountain [Fountain83] compare SIMD implementations. Gerritsen considers performance and configuration parameters such as cycle counts, clock speed, number of PEs, and I/O architecture. Fountain examines register structure, logic diagrams, and other details of PE implementation. This level of detail is appropriate for system designers and architects, rather than programmers and language designers.

In a similar vein, Kohonen [Kohonen80, chapter 6] covers the intersection of CAM (Content Addressable Memory) and SIMD architectures in great detail. He makes no distinction between the details normally hidden from programmers by a compiler, and the higher level features which are important to algorithm selection.

An extended abstract by Jamieson [Jamieson87] considers matching algorithms with all kinds of parallel architectures, not just SIMD. Karp [Karp87] presents a taxonomy restricted to "those aspects that affect coding style," but considers only MIMD (Multiple-Instruction Multiple-Data) architectures.

Though each of these earlier taxonomies has its own purpose and merit, none is an appropriate tool for designing an optimally portable language.

The taxonomy presented in this chapter begins with a precise definition of equivalence. This not only leads to an operational rule for distinguishing architectures and their features, but it also brings to the surface more dimensions along which existing architectures differ. The result is a larger number of descriptors, which makes it possible to more nearly reconstruct the properties of an architecture from its classification.

Chapter 4

Portability of Existing SIMD Languages

Like any other family of computer architectures, software tools are necessary to harness the power of SIMD computers. The most important such tools are appropriate programming languages. Unlike vector processors, where language research has focused largely on discovering implicit parallelism in existing sequential programs, efforts for SIMD machines have focused on new languages in which parallel data operations can be expressed directly. This reflects the consensus that utilizing the much greater parallelism of SIMD machines requires new and different algorithms not found in existing sequential programs, and that these parallel algorithms are best expressed in appropriate parallel languages.

Whereas not all research projects involving SIMD architectures have advanced to the point of developing language tools, some have. The result is a variety of high- and low-level languages for expressing data-parallel, or SIMD, algorithms. One striking characteristic of this body of language research is that almost all efforts have been part of a larger effort to develop and use a particular SIMD architecture. There have been very few independent projects to develop languages for SIMD computers in general.

One reason for this is that SIMD computers exhibit much greater architectural diversity than sequential computers, which essentially all share the same von Neumann architecture. The level of abstraction common among programming languages is sufficient to hide the minor architectural differences among sequential architectures, making it possible to implement most sequential languages efficiently on virtually all sequential computers. The SIMD taxonomy presented in the previous chapter shows that this is not true for SIMD languages and architectures.

Language designers, who have been concerned primarily with supporting some particular SIMD architecture, have handled this problem by designing a language suitable for the particular set of features provided by that architecture. As a result, each existing SIMD language contains architectural assumptions which make it suitable for programming only a certain subset of SIMD machines: those which closely resemble the machine for which it was designed.

Optimal portability solves this problem. Its definition provides a mechanism for accommodating architectural diversity within a single language, and provides a criterion for identifying the architectural distinctions the programmer should be allowed to see. An optimally portable language requires each program to specify its target architecture, i.e., its architectural assumptions and precise portability. An optimally portable language allows each program to use all the language features that are supported by the program's target architecture; but it does not allow the program to use any language feature not supported by that target architecture.

4.1 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 chapter. Some languages are not portable at all. To my knowledge, Fortran 8X is the only language that has been implemented in any form on more than one SIMD machine. In the brief survey of SIMD languages below, most languages are grouped by machines. Very low-level languages are not considered, leaving no languages to discuss for some machines.

4.1.1 Illiac IV Languages

Three main languages were developed for the Illiac IV: GLYPNIR (Algol-like), CFD (Fortran-based), and IVTRAN (Fortran-based). [Hord82] 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 [Perrott75] was also developed. Actus is limited by its assumption of 2D grid communication.

4.1.2 MPP Language

As defined, Parallel Pascal is suitable only for architectures with a 2-dimensional rectangular inter-PE communication network. The MPP's implementation of Parallel Pascal also fails to insulate programmers from hardware details, contrary to the language definition. [Potter85]

4.1.3 BVM Language

BVL-0 (Boolean Vector Language 0) [TombouliMW85, Tuck87] 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.

4.1.4 BSP Language

The BSP Fortran Vectorizer [HwangBrigg84, 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.

4.1.5 MP-1 Language

MPL (MasPar Parallel Application Language) assumes the presence of all the MP-1's important architectural features. These include adjacent, piped and global communication, local addressing, and reduce and scan operations. [MasPar Com90b, MasPar Com90a]

4.1.6 CM Languages

C*, CM-Lisp, and *Lisp all assume the presence of the Connection Machine's support of global communication, local addressing, and local addressing during communication. Some also assume the presence of reduce and scan operations. [Thinking M87b, RoseSteele87, SteeleHill86, Thinking M88] Paralation Lisp assumes at least support for global communication with completely arbitrary combining functions for collision resolution during write, which is not supported by any architecture classified in chapter 3, and replication on fetch collisions. It also assumes support for scan operations. [Sabot87, Sabot88a, Sabot88b] CM++, which has both machine-independent and machine-dependent aspects, assumes the full power of the Connection Machine architecture. [Collins90]

4.1.7 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 [Massachuse84], but not implemented [AlbertKnLS88]. More recently, a subset of Fortran 77, with proposed Fortran 8x array extensions (including some "removed extensions"), has been implemented for the CM [AlbertKnLS88]. MasPar has announced its intention to deliver a very similar product for the MP-1. This reflects the many architectural similarities of these machines, evident in their classifications in the previous chapter. FORTRAN-PLUS for the DAP 500 is an implementation of Fortran 77, minus I/O facilities, plus some proposed Fortran 8x array extensions [ParkinsoHM88, Active Mem87b]. There are significant differences in the languages supported for the CM and DAP. These reflect their fundamental architectural differences.

The proposed Fortran 8x standard [Technical87] 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 2-D grid communication heavily, so it cannot be implemented on all SIMD architectures.

4.1.8 Other Languages

Hamet and Dorband propose a new language which they describe as "a generic fine-grained parallel C" but do not explicitly name. [HametDorba88] Its parallel control structures assume support for global "or" reduction, and its interprocessor communication statement assumes support for unrestricted piped 1-D adjacent communication. Most other optional architectural features are not provided, but there are plans to add unlimited local addressing.

Miller and Stout propose an approach to language portability based on common communication operations. It is embodied in the language Seymour. [MillerStou88b, MillerStou88a] These operations include scanning, reduction, global communication, and sorting.

Rice, Seidman, and Wang present a SIMD language amenable to proofs of program correctness. [RiceSeidmW90] It explicitly assumes a hypercube (N-D) topology with adjacent communication. It does not provide local addressing or reduction.

4.2 Existing Languages Are Not Optimally Portable

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 allows the use of one or more features not present in all architectures, and most fail to allow the use of some feature present in some architecture. Therefore, they all fail to satisfy the second or third part of the definition of optimal portability.

4.3 Non-Procedural Languages

The research reported in this dissertation 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. There has been relatively little work on non-procedural languages specifically for SIMD architectures. CM-Lisp or Paralation Lisp (section 4.1.6 might be considered non-procedural.

O'Donnell [Odonnell88] discusses the SIMD architectural features required to efficiently implement various operations typical of declarative programming languages. This tends to confirm that in declarative languages, as in procedural languages, each feature of a language has its own architectural assumptions.

Chapter 5

Porta-SIMD Language Design

Previous chapters have defined optimal portability and described its benefits. They presented a taxonomy suitable for evaluating optimal portability for SIMD architectures, and concluded that no previous SIMD language is optimally portable. This chapter shows that optimal portability is an achievable goal for SIMD language design. It shows this by presenting the design of a new language, Porta-SIMD.

The organization of this chapter is as follows. First, the requirements an optimally portable language must meet are discussed in greater detail than in the preceding chapters. Then the major influences on Porta-SIMD's design are discussed. These include the strategy for meeting the requirements of optimal portability, the choice of implementation technology and its impact on the language design, and secondary design goals. A brief overview of the language and some small examples give a feel for the language before section 5.4 presents the Porta-SIMD language definition. Finally, some potential future enhancements are discussed.

5.1 Requirements for an Optimally Portable Language

The definition of an optimally portable language in chapter 2 has three requirements. The language must make each program *specify* a target architecture, it must *provide* all the features of that architecture, and it must *enforce* a ban on language features the target architecture does not support. The SIMD taxonomy presented in chapter 3 is tailor-made for specifying target SIMD architectures and the features they support, doing so in a way that satisfies the supporting definitions of optimal portability. The rest of this discussion assumes architectures are specified using chapter 3's taxonomy.

5.1.1 Specify Target Architecture

The requirement that each program specify its target architecture can be met in several ways. The target architecture may be implicitly specified, or it may be explicitly specified at several levels. Here are some examples.

Program level The target architecture is specified at program link time, perhaps with a linker option or a library name.

File level A target architecture for each source code file is specified within the file. The compiler and linker combine these specifications to obtain the overall target architecture, which must subsume those of the files.

Algorithm level A target architecture for each algorithm is specified within the source code for the algorithm. The compiler and linker combine these to obtain the overall target architecture.

Each of these explicit specification levels has a corresponding enforcement level.

Program level Verify at link time that no file being linked uses a feature not supported by the target architecture.

File level Verify when compiling each file that it uses only features supported by its target architecture.

Algorithm level Verify when compiling each statement and operation that it uses only features supported by the target architecture specification applicable to that statement and operation.

These explicit specification and enforcement levels have different advantages and disadvantages for programmers. The program level makes the program's portability most clear. The programmer specifies an overall target architecture, and the language implementation verifies that the program will run efficiently on that target architecture. However, programs are usually composed of many algorithms, and those algorithms often use different sets of architectural features. Reusing algorithms in different programs is easiest if each algorithm's minimum target architecture is specified as part of the algorithm's expression in the language. Separating such an important characteristic from the algorithm's source code and putting only an overall target architecture in the program's makefile increases the potential for programmer error. It makes it much easier to accidentally use an algorithm which requires features not present in the program's target architecture, and postpones recognition of the problem until the program is linked. Using file level target architecture specification finds such problems earlier, but still separates an important algorithm characteristic from the algorithm's source code if the file contains more than one algorithm.

Using only implicit specification of the target architecture is essentially the same as using only program level specification, and specifying as the target architecture the architecture of the SIMD computer for which the program is currently being compiled. When explicit specification is used at some, but not all, levels, the effect is to use implicit specification at the remaining levels.

I believe the programmer is best served by using all three levels of explicit specification and enforcement. Each algorithm's expression in the language must include a specification of its target architecture. This makes the algorithm and its architectural assumptions inseparable and aids correct algorithm reuse. The target architecture specification in each file and for the program as a whole serve two purposes. They summarize the architectural assumptions of the program and its parts, and they are assertions about the portability of the program and its parts. A summary which understates portability by specifying spurious architectural assumptions is acceptable, and perhaps occasionally useful. (However, a compiler might usefully warn the programmer whenever an architectural feature is specified at any level but not used within the scope of the specification.) But a summary which overstates portability by omitting architectural assumptions is incorrect; its assertion fails, and it evokes an error message. Porta-SIMD provides all three levels of specification and enforcement.

There are several ways of integrating algorithm level target architecture specification into a language. The method used by Porta-SIMD augments the notion of data type to include an architecture specification. Then it requires that every operation be supported by the shared architecture of the types of its operands.

I considered two alternative methods of algorithm level target architecture specification. The first specifies an architecture for each subroutine as part of its declaration. This lets the architecture be stated in fewer places. Although a subroutine may contain more than one algorithm, a subroutine is a common and useful unit of code for reuse. The second alternative allows any block scope to be labeled with an architecture, and requires every parallel operation to be within such a labeled scope. Further, nested scopes are allowed, but a nested scope may only be labeled with an architecture subsumed by the enclosing scope's label. That is, an architecture scope may restrict the architecture of an enclosing scope, but may never specify an architectural feature not supported by the enclosing scope. An architecture scope is not allowed to be larger than a single subroutine, so it cannot degenerate into file level target architecture specification. These two alternatives can be combined, and the combination is probably better than either alone.

Both Porta-SIMD's type-based method and the alternative scope-based methods adequately support algorithm level specification of architectural assumptions. Implementation technology played a large role in the decision to use type-based architecture specification in Porta-SIMD.

5.1.2 Provide Architectural Features

An optimally portable language must be able to provide access to every architectural feature that can be part of any target architecture. That is part of the definition of optimal portability.

The SIMD taxonomy presented in chapter 3 defines 12 categories containing a total of 94 architectural features (if fixed-dimension cartesian coordinates with more than two dimensions are temporarily ignored). However, some of these features modify the behavior of other features when both are present in an architecture. So providing all the required features includes providing them in all their various modified forms.

5.1.3 Enforce Architectural Limits

An optimally portable language must be able to prevent a program from using any language feature that provides an architectural feature which is not part of the target architecture. The interaction of architectural features means that some forms of a feature may only be used when another feature is present. This complicates the implementation of enforcement somewhat.

5.2 Design Strategy and Goals

This section introduces some important aspects of Porta-SIMD's design. I decided very early to do a prototype, or "proof of concept," implementation without writing a compiler. Instead, I used C++ features to define new parallel data types and overload the standard arithmetic operators to work with these new types. This decision and the influence it had on the design of Porta-SIMD are discussed in the first subsection. The second subsection describes the strategy Porta-SIMD uses for satisfying the requirements of optimal portability. The final subsection discusses language features and design goals not directly related to optimal portability.

5.2.1 Implementation Technology

The purpose of designing Porta-SIMD was to demonstrate that optimal portability is an achievable goal for SIMD languages. To bolster this argument, I decided to build a prototype implementation of Porta-SIMD. An important purpose of this prototype is to show that the language design is complete and detailed enough to allow implementation. Another was to provide feedback to the language design by developing the prototype concurrently with the language.

Writing a compiler for Porta-SIMD was not the best way to accomplish these goals. For me to write a compiler in a reasonable amount of time, working alone, would have required designing a very stripped-down toy language. It would have been difficult to experiment with the language early on, and difficult to change it based on that experience.

Using C++ [Lippman89, Stroustrup89, Stroustrup86] as the base language and implementation tool provided an excellent alternative. I used its class definition and operator overloading capabilities, and its preprocessor macros, to define parallel data types and make them appear very much like they were built in. Implementing Porta-SIMD entirely as C++ `#include` files and libraries let me begin with a tiny prototype and grow it through several versions to its final form. Experience with each version helped shape the language design and the next version.

C++ also provided a full, well-known base sequential language to extend. There was no need to invent new syntax and semantics for the sequential sections of SIMD programs, or to spend time implementing them. And a lot was gained by using a language with which programmers were already familiar. Using C++ was also a good choice for implementation portability, because it was available on all the sequential hosts I used. It was only necessary to port the machine-dependent parts of my class implementations.

However, C++ has limits that influenced Porta-SIMD's design and implementation. The implementation limits are discussed in the next chapter. The primary influence on language design was the facilities C++ provides, and does not provide, for "extending" the language. C++ class definitions and operator overloading let me define SIMD parallel data types and their use in C++ expressions. However, C++ provides no mechanism for adding new attributes of data types. Therefore, the parallel data types have information encoded in their type names which would be better expressed as type attributes. Being parallel is the first such attribute. Another is the set of architectural assumptions the type makes. These attributes are encoded in the `simd_ARCHID` part of the type names, as described in detail in section 5.4.4.2.

C++ does not provide a mechanism for extending the semantics of control structures. So Porta-SIMD's parallel flow control is done with macros. They are functional, but not as convenient or attractive syntactically as I would like. There are also some restrictions on the use of casts. These are an artifact of the way user-defined type conversions may be specified. Fortunately, this does not restrict the function of the language, because conversion by assignment to an explicitly declared temporary variable is always possible.

C++ had a more important influence on the strategy used to achieve optimal portability in Porta-SIMD. This is discussed in the next subsection.

Overall, the benefits of using C++ as the language base and implementation tool for Porta-SIMD are substantial. The limitations inherent in this approach were acceptable, and more than balanced by the benefits.

5.2.2 Strategy for Optimal Portability

Porta-SIMD provides all three levels of target architecture specification and enforcement: program, file, and algorithm. The way this is done fits very well with the C++ implementation strategy, and takes full advantage of C++'s strong type checking. Porta-SIMD provides access to all features in the SIMD taxonomy.

The program level target architecture must be specified by linking with the library that is named for the target architecture. Exactly one such library must be used, and the libraries are such that this is enforced by the linker.

The file level target architecture is specified by including the file of Porta-SIMD definitions named for the target architecture. The C++ compiler enforces the architectural restrictions of this target architecture, because the `#include` file only defines the data types and operations supported by the specified target architecture. The `#include` files verify that only one target architecture is specified within any file.

The algorithm level target architecture is specified using data types. Each parallel data type includes an architecture specification as part of its name. The operations defined for each data type include only those supported by its architecture. This approach uses the strong type checking of C++ to fulfill optimal portability's enforcement requirement.

Porta-SIMD provides basic PE computation by defining the basic C++ operators for all parallel data types. It also provides parallel "if-then-else" flow control and a few other basic operations for all SIMD architectures. Porta-SIMD provides optional architectural features appropriate for each type's architecture. These are provided primarily in the form of named member functions. A few are also provided via overloaded operators, but there are too many operations and variations of similar operations to provide more than a few of them through operator syntax. Of course, not all operations should be expressed with operators. Some operations have so many variations that expressing them with operators may cause more confusion than benefit. C++ does not allow new operators to be added, though it is not clear whether more operators would be a good idea. There is room for future improvement in this part of Porta-SIMD's design.

5.2.3 Other Goals

My primary goal in designing Porta-SIMD was to convince observers that it is possible to design a real, full-featured optimally portable SIMD language. This does not require designing the perfect such language, but it does require the language design to provide optimal portability while also dealing successfully with some other important issues. I did not design a "toy" language stripped of all features not directly related to optimal portability, because I want to show that optimal portability is a practical and useful design goal for real commercial SIMD languages.

As a result of this goal, Porta-SIMD provides better features than C* and MPL in three important areas unrelated to optimal portability. These are virtual machines, allocation of PE memory, and generic parallel subroutines. Porta-SIMD provides multiple virtual SIMD machines on which to allocate parallel variables. Each virtual machine has its own size and shape, which need not be the same as the underlying hardware machine. Each variable is associated with a virtual machine, and has one element stored in each of that machine's (virtual) PEs. The primary purpose of the virtual machine is to specify the size and shape of its variables. It also ensures that its variables have the same size and shape, and may therefore be operated on together. Virtual machines may be allocated dynamically during program execution, with dynamically computed size and shape. MPL intentionally does not provide virtual machines. C*'s domains and domain arrays provide something similar to statically allocated virtual machines, but with some characteristics I find odd. Domains are like virtual machines in the sense that parallel variables are allocated within domains, and domains are used to activate sets of PEs and operate on the domain's variables. But each domain is a type (like a structure name), not a variable. And the size and shape are specified by declaring arrays of some domain type. These arrays can only be allocated statically.

Porta-SIMD allows parallel variables of all three important storage classes: global (static), local (automatic), and dynamic. MPL also provides all three storage classes, but C* does not provide dynamic allocation of domain (parallel) variables.

A generic parallel subroutine is able to perform some computation on any virtual machine specified at run time. Porta-SIMD allows parallel variables to be allocated on any virtual machine; it allows virtual machines to be passed as subroutine arguments like any other data type; and it provides a means of determining which virtual machine any parallel variable is on. These features allow a Porta-SIMD subroutine to allocate parallel (local or dynamic) variables on the same virtual machine on which a parallel argument was allocated, perform its computation on that virtual machine, and even return a parallel variable allocated on that machine. If several parallel arguments are on different virtual machines, there are globally declared virtual machines, or there are locally declared virtual machines, the subroutine can use any or all of these virtual machines. This is especially useful for writing utility routines for libraries, since very little is known about the application calling such library routines. Because C* activates sets of PEs based on the domain name, which is a type, a C* routine must specify at compile time the set of PEs it will run on. There is a global "all PEs" domain, but most variables are not accessible from it. Generic parallel subroutines are not applicable to MPL, since it provides only a single real machine.

5.3 Language Overview and Example

Porta-SIMD extends C++ by adding SIMD parallel data types. Most C++ operators are defined for these new types, and some additional operations are defined as C++ "member" functions of the parallel data types. C++ is very nearly a strict superset of C [HarbisonSt84, KernighanR78]. Users familiar with C, even if they are not familiar with C++, should have little trouble using Porta-SIMD. Porta-SIMD's extensions to C++ can be summarized as follows.

Architecture Identifiers Each legal architecture specifier defined by chapter 3's SIMD taxonomy can be used as an *architecture identifier* in Porta-SIMD. This includes the basic SIMD architecture with no optional features, denoted by the null string.

Architecture identifiers are used within the names of parallel data types, as described in the following paragraphs. When a type name that may contain any architecture identifer is written, the place for the architecture identifier is marked `ARCHID`. If the null architecture identifier is used and would cause two underscore (_) characters to appear together, only one underscore is written.

Virtual SIMD Machines Porta-SIMD provides virtual SIMD machines of all architectures as data types. The types are named `simd_ARCHID_mach`. Each virtual machine may be declared with any size (and shape, if `ARCHID` contains a cartesian labeling). Providing multiple virtual machines allow a program to use parallel variables with a variety of sizes and shapes.

Parallel Data Types For each integer and floating-point data type in C++, there is a corresponding SIMD parallel type consisting of a data element of that type in each PE of a virtual SIMD machine. This parallel data type is named by prepending `simd_ARCHID_` to the name of the C++ data type. For instance, `simd_int` is the parallel version of the type `int` for the basic SIMD architecture. Most C++ operators are defined for these parallel data types, and parallel expressions are evaluated in each enabled PE.

Optional Architectural Features Porta-SIMD provides C++ member functions, or methods, for each parallel data type. These methods provide access to the optional architectural features specified by the type's `ARCHID`.

Flexible Data Type Sizes Each of the `simd_` integer data types may be declared to contain any number of bits from 2 to some implementation-defined maximum. In addition, a boolean `simd_ARCHID_bool` type is provided which requires only a single bit in each PE. These extensions allow better use of the limited local PE memory.

Extended Type Conversion Rules The C++ rules governing type conversions within an expression containing multiple data types have been extended to encompass the new parallel types.

Flow Control A parallel "if-then-else" conditional execution mechanism is provided. For architectures which support it, there is also a parallel "while" statement.

Porta-SIMD is described fully in the next section, but a couple of examples are used here to show the flavor of the language. The first example, shown in figure 5.1, is a trivial graphics program that displays a single rectangle and requires the rectangle to be aligned with the coordinate axes by which the PEs are labeled. It begins by declaring with an `#include` statement that its architecture identifier is `2dO`, meaning it assumes that the PEs have a 2-D labeling and that a display device is supported by the architecture. It also declares a global virtual machine with this architecture and default size.

The `rectangle()` function returns a boolean flag in each PE of the virtual machine, indicating whether the PE is in the rectangle defined by the function arguments. It does this by declaring and initializing variables which hold the coordinates of each PE, then comparing these coordinate variables to the arguments and accumulating the results of the comparisons. The main program simply calls `rectangle()` and displays the result.

The second example, shown in figure 5.2, is a slightly modified fragment of a "Game of Life" program written by Greg Turk. The entire program is presented in section 6.2.5. As every Porta-SIMD source file must, this one specifies the file's architecture identifier before using any parallel features. In this case, the identifier is `2aCG`, meaning that this code requires 2-D adjacent communication and full PE to host communication on the target architecture.

The remainder of figure 5.2 is a routine which computes a generation of the game of life, as follows. Given a 2-D array of cells, one per PE, and the initial state of the cells (whether each is occupied or not), it uses the following rules to compute the next state of each cell. An occupied cell remains occupied if either two or three of its eight adjacent cells are occupied, otherwise it becomes unoccupied. An unoccupied cell becomes occupied if exactly three of its eight adjacent cells are occupied, otherwise it remains unoccupied.

This example introduces three important language features. First, the subroutine `compute_iterations()` will run on any virtual machine specified at run time. The virtual machine is specified implicitly by the `cell` argument, which exists on some virtual machine. The routine declares each local parallel variable to be on the same virtual machine as `cell`, using the `m_2aC()` method to query `cell` for its machine. I call a routine such as this which may be used on any virtual machine specified by its arguments a *generic* parallel routine. I think the ability to write generic parallel routines is an important and powerful language feature. In particular, it is difficult to write readily reusable code when important information such as which virtual machine to use is communicated implicitly by common assumptions or global variables.

Some important SIMD languages, including C* [Thinking M87b, RoseSteele87] and CM-Fortran [AlbertKnLS88], do not support generic parallel routines. CM-Lisp [SteeleHill86] and Paralation Lisp [Sabot87, Sabot88a, Sabot88b] do support them, as do the machine-specific languages Paris [Thinking M87b, Thinking M89] and CM++ [Collins90]. *Lisp [Thinking M87b, Thinking M88] provides sufficient but minimal support: virtual machines may be declared and passed as subroutine arguments, and parallel variables may be allocated on such a virtual machine, but there is no way to determine dynamically which virtual machine an existing parallel variable is on. MPL's [MasPar Com90b, MasPar Com90a] single real PE array model does not support generic parallel routines.

```
#include <simd_2d0.h>

/* Accept the coordinates of the upper left and lower right
 * corners of a rectangle with sides parallel to the axes.
 * Return 1 in each PE inside the rectangle, 0 in those outside.
 */
simd_2_bool rectangle(simd_2_mach *m,
                      int x1, int y1, int x2, int y2)
{
  simd_2_bool inside(m);
  simd_2_int x(m, 16), y(m, 16);
  x.pe_coord(dimen_x);
  y.pe_coord(dimen_y);
  inside = 1;
  inside &= (x > x1);
  inside &= (y > y1);
  inside &= (x < x2);
  inside &= (y < y2);
  return(inside);
}

main()
{
  simd_2d0_mach mach;
  simd_2d0_bool result(&mach);
  result = rectangle(2,6,24,57);
  result.display();
}
```

Figure 5.1: rectangle.C — Example Porta-SIMD program.

The second feature introduced by this example is adjacent communication. Four direction constants are declared, named for the compass directions. These have type `direction_aC`, meaning they specify a dimension along which to communicate, and a direction along that dimension. The `fetch()` method operates in each enabled PE by fetching data from the adjacent PE in the direction specified by its second argument (`N`, `S`, `E`, or `W` in this example). The data in each remote PE is taken from the variable named by the first argument (`cell` or `tsum`), and is stored in each fetching PE in the variable on which `fetch()` is invoked (`tmp`).

The third feature introduced by figure 5.2 is the parallel `IF-ELSE-ENDIF` statement. Although sequential "if" statements control the flow of execution, that is not precisely the case for SIMD parallel "if" statements. Instead of using a scalar test expression to decide *whether* to execute the body of a sequential "if" statement, a parallel "if" uses a parallel test expression to decide *in which* PEs to execute the body. In particular, the code following each of the `IF` and `ELSE` keywords is executed, but only in the appropriate (and mutually exclusive) sets of PEs.

```
#include <simd_2aCG.h>
/* Compute several generations of life.
 *   cell  -- life grid of cells
 *   iters -- number of generations to compute
 *            (number of times to advance the state)
 * Does all computation on virtual machine holding "cell".
 */
void compute_iterations(simd_2aCG_bool& cell, int iters)
{
  int i;
  const direction_aC E(dimen_x,1), W(dimen_x,-1),
                     N(dimen_y,1), S(dimen_y,-1);
  simd_2aC_int tsum(cell.m_2aC());
  simd_2aC_int sum(cell.m_2aC());
  simd_2aC_int tmp(cell.m_2aC()); /* used in communication */

  for (i = 0; i < iters; i++) {

    /* count the living neighbors */
    /* tsum = self + NS neighbors (3 cells) */
    tsum  = cell;
    tsum += tmp.fetch(cell, N);
    tsum += tmp.fetch(cell, S);

    /* sum = tsums of self and EW neighbors (9 cells). */
    sum   = tsum;
    sum  += tmp.fetch(tsum, E);
    sum  += tmp.fetch(tsum, W);
    /* subtract self from sum, so have 8 neighbor count. */
    sum  -= cell;

    /* compute new generation */
    IF (cell)
      IF (sum < 2 || sum > 3)
        cell = 0;
      ENDIF
    ELSE
      IF (sum == 3)
        cell = 1;
      ENDIF
    ENDIF
  }
}
```

Figure 5.2: Fragment of example Porta-SIMD “life” program.

5.4 Porta-SIMD Language Definition

This section defines Porta-SIMD's extensions to C++ [Lippman89, Stroustrup89, Stroustrup86]. These extensions have been made as much in the spirit of C++ as possible, and in most cases existing language rules apply to the extensions as well. These cases will generally not be mentioned, in order to focus on the extensions themselves and on the few exceptions.

The initial sections of this language definition describe the base language defined on all SIMD architectures. These are followed by sections describing language features which provide access to optional SIMD architectural features.

Porta-SIMD defines a variety of types, with a variety of declaration arguments. Besides the declaration arguments described with each such type, every type Porta-SIMD defines may also be declared with a single argument, which is a value of the same type. The declared object is initialized to the value of the argument. Each type Porta-SIMD defines also includes the assignment operator, so values of that type may be assigned to lvalues of that type.

The prototype Porta-SIMD implementation does not implement the entire language. It implements the integer data types, but not floating point types. It implements only selected optional architectural features. The prototype implementation is described in detail in the next chapter.

This language definition is intended to be complete. However, if it is ambiguous or incomplete in some way, it may be helpful to consult the public interface specifications of Porta-SIMD's parallel data types, in appendix A.

5.4.1 Reserved Words

The identifier `simd` and all identifiers beginning with the prefix `simd_` are reserved by Porta-SIMD and may not be used as program identifiers. The prefixes `direction_` and `label_` and the identifier `shape` are likewise reserved. The following keywords used in parallel flow-control constructs are reserved: `ALL`, `ENDALL`, `IF`, `ELSE`, `ENDIF`, `WHILE`, `ENDWHILE`.

5.4.2 Architecture Identifiers

The architecture classification names defined by the taxonomy presented in chapter 3 are used as architecture identifiers. An architecture identifier therefore identifies a particular set of optional architectural features, which defines a SIMD architecture. The null architecture identifier, an empty string, identifies the base SIMD architecture with no optional features.

Architecture identifiers are used within the names of parallel data types. When a type name that may contain any architecture identifer is written, the place for the architecture identifier is marked `ARCHID`. If the null architecture identifier is used and would cause two underscore (_) characters to appear together, only one underscore is written.

One architecture identifier is said to be a subset of another if its architecture is subsumed by the other's architecture. (I.e., an architecture which can be simulated with a constant-bounded number of operations and data elements is a subset of the simulating architecture.) A program may, through its data declarations, use any number of architecture identifiers, provided they are all subsets of some *overall architecture identifier*. There may be several such overall architecture identifiers, provided they are all equivalent. (Architecture equivalence is defined in chapter 2.)

5.4.3 Target Architecture

Every Porta-SIMD source file must specify its target architecture by including exactly one file whose name has the form `simd_ARCHID.h`. (If the null architecture identifier is used, the underscore is omitted.) This must be done before any parallel data type is named in the file. No architecture identifier may be used in the file unless it is a subset of the file's target architecture.

Every Porta-SIMD program must specify its overall target architecture during the link phase of its compilation. The method used to specify the overall target architecture is implementation dependent. No architecture identifier may be used in the program unless it is a subset of the program's overall architecture identifier.

5.4.4 Types

Porta-SIMD extends C++ with two new sets of types: virtual machine types and parallel data types. Objects of these types can be statically declared and dynamically allocated.

5.4.4.1 Virtual Machines

Porta-SIMD allows a program to perform parallel computations on multiple, logically distinct, virtual SIMD machines. A type name of the form `simd_ARCHID_mach` is used to declare a virtual machine with any architecture identifier.

Having multiple virtual machines with different sizes and shapes allows a program to operate on different kinds of data with different natural sizes and shapes. When the implementation supports virtual processor to hardware processor ratios greater than one, multiple virtual machines are even more important because they allow the best virtual processor ratio to be used for each kind of data.

A single default virtual machine is provided by Porta-SIMD without explicit declaration within the program, and has the program's overall architecture identifier. It has no name, but is used implicitly in certain parallel data declarations as described in section 5.4.4.3.2. Any other virtual machines used by the program must be explicitly declared before their use.

5.4.4.2 Parallel Data

Porta-SIMD provides parallel integer, floating point, and boolean data types. Every parallel object is associated with a virtual SIMD machine, and consists of a set of identical elements, one per PE of its virtual computer. Parallel data type names have the form `simd_ARCHID_TYPE`, where `TYPE` is one of: `int`, `unsigned`, `float`, `double`, `bool`. The choice used for `TYPE` is called the *basic data type*, or just basic type.

The parallel integer `int` and `unsigned` types satisfy the same C++ language rules as their sequential counterparts. However, they differ in one way. The amount of PE storage used to represent these integers may be specified when they are declared or allocated. (See section 5.4.4.3.) Therefore, the C++ types `short` and `char` have no parallel counterparts; appropriate parallel `int` and `unsigned` declarations replace them.

The parallel floating point `float` and `double` types are just like their sequential counterparts in C++. The implementation of parallel floating point types may be different than the implementation of sequential floating point types, but both must satisfy the same C++ language requirements. If the parallel and sequential implementations are not identical, conversion between parallel and sequential floating point types must be done correctly by the implementation (see section 5.4.6.1).

The parallel `bool` type holds a boolean value in each PE.

Because their names begin with simd_, parallel types are also called simd_ types. The term simd_ types does not include the simd_ARCHID_mach types, despite their similar name. They are referred to as the virtual machine types, or simply machine types. The int and unsigned simd_ types are called the *integer* simd_ types. The float and double simd_ types are called the *floating point* simd_ types. The integer and floating point simd_ types are together called the *numeric* simd_ types.

simd_ types containing the null architecture identifier are called "base" simd_ types, and are available on all SIMD computers. Therefore, operations are provided for base simd_ types only if they are supported by all SIMD architectures. The remaining simd_ types are called "derived" simd_ types. Each derived simd_ type provides all the operations of its corresponding base type, plus additional operations which take advantage of the optional architectural features specified by its architecture identifier.

5.4.4.3 Declaration

5.4.4.3.1 Virtual Machine Declarations Virtual machines may be declared with or without arguments. Without arguments, the virtual machine has an implementation defined size and shape, which is usually the same as the hardware machine in use. With arguments, the virtual machine has the size and shape they specify. The number, type, and meaning of the arguments depends on the labeling and communication features of the architecture identifier.

The base machine type, simd_mach, has one argument: the number of PEs. Other virtual machine types have the same argument as the base type, unless they contain one of the following features.

1 One argument: the number of PEs.

2 Two arguments: the number of PEs in each dimension.

... ... arguments: the number of PEs in each dimension.

N One argument of type shape, which specifies the number of dimensions and the number of PEs in each dimension. The type shape is described in a succeeding paragraph.

p2 One argument: the number of levels in the pyramid.

cC One argument: the number of PEs; however, the implementation may provide a virtual machine with more PEs. This allows it to provide a full CCC network, or a convenient portion of one.

An argument value of zero specifies that the default value be used. The result of declaring a virtual machine with arguments the implementation cannot satisfy (e.g., requested too large a machine) is undefined.

The type shape is provided in source files whose target architecture includes the "N" labeling. Its value is used to specify the size and shape of a virtual machine with the "N" labeling, when the machine is created. (The shape type could be provided for architectures with fixed-dimension cartesian labelings, but a run-time check would sometimes be required to verify that a shape had the right number of dimensions.) A shape object may be declared in either of two ways: (1) a single unsigned scalar argument specifies the number of dimensions; (2) a single shape argument initializes the new object to the argument's value. The operator [] may be applied to a shape lvalue. It has a single unsigned scalar argument which specifies a dimension of the lvalue. The operator [] returns a reference to that dimension's length (in PEs), so it may be used to set or examine the value of the shape. The length of each dimension of a shape value must be initialized before the value can be used, either individually with [], or collectively by initializing with another shape value. The dimens() method may be applied to any shape value. It has no arguments, and returns the number of dimensions in the shape value.

Here are some example declarations.

```
simd_mach m, m_a(4096);
int i = 32;
simd_gCaWrFgsRgsSIO_mach m_big(i * 1024);
simd_2adwCucPIOG_mach m_rect_a;
simd_2adwCucPIOG_mach m_rect_b(i, 64);

shape cube_shape(3);
cube_shape[0] = 32;
cube_shape[1] = 32;
cube_shape[2] = 32;
simd_NgCIO_mach m_cube(cube_shape);

simd_p2mCG_mach m_pyr(5);
simd_cC_mach m_ccc_a, m_ccc_b(2048);
```

5.4.4.3.2 Parallel Data Declarations Objects of the simd_ types may be declared with or without arguments. Without arguments, they are placed on the default machine and have the default size. The default machine has the program's overall architecture identifier and an implementation dependent size (usually the size of the hardware).

The arguments which may be specified in the declaration of a simd_ variable depend on its basic data type. For all basic types, the first argument is a pointer to a virtual machine. The object is created on the virtual machine pointed to. The architecture identifier in the declaration must be a subset of the architecture identifier of the virtual machine pointed to. (The program is aborted with a message if this is not true.) If the argument value is a null pointer, or no arguments are specified, the default machine is used. If the declaration is for an integer basic type, there is a second argument discussed in the next paragraph. For other basic types, there is only one argument.

The "length" of a simd_ type is the number of bits it is declared to contain in each element. The length of an integer simd_ type can be specified as the second argument to its constructor. This length may be any integer from 2 to some implementation-defined maximum, which must be at least large enough to store all possible values of a scalar C++ int in that implementation. If no length, or a length of zero, is specified in the declaration of an integer simd_ type, an implementation defined default length is used. (The default length should be at least 16 bits, the minimum default sequential int size in existing C++ implementations.) Here are some example declarations.

```
simd_unsigned u_a, u_b(&m_big, 8);
simd_int tiny_i(NULL, 2), larger_i(NULL, 32);
simd_2aCIOG_unsigned u_c(&m_rect_a);
simd_2aCIOG_unsigned u_d(&m_rect_b, 0);
```

The length of boolean simd_ types is always 1.

```
simd_bool b0, b1(&m_a);
simd_2adCgoR_bool b_2d;
```

The lengths of floating point simd_ types are implementation dependent.

```
simd_float f, fprime(&m_cube);
simd_gCgoRIOG_double g;
```

5.4.4.4 Allocation

A simd_ value is not a single value stored in the host computer, but a set of values stored one per PE. Similarly, virtual machines represent portions of physical SIMD machines. For this reason, simd_ objects and virtual machines may not be stored in malloc()'d memory. Instead, dynamic allocation is performed using the C++ new operator, and anything so allocated is freed with the C++ delete operator.

```
simd_int *sipa;
simd_gCIO_int *sipb;
sipa = new simd_int;
sipb = new simd_gCIO_int(mbig, 17);
delete sipa;
delete sipb;

simd_mach *mp;
mp   = new simd_mach(4096);
sipa = new simd_int(mp, 0);
delete sipa;
delete mp;
```

A program is aborted with an error if it uses in any way a parallel data object residing on a virtual machine which does not exist (e.g., has been deleted). It is therefore wise to ensure that all simd_ objects residing on a virtual machine have been deleted or gone out of scope before the virtual machine is deleted or allowed to go out of scope.

5.4.4.5 Lvalues

As in C++, an lvalue is an expression referring to a region of storage (an object). Such an expression is called an "lvalue" because it may appear on the left side of an assignment expression. Non-lvalue expressions are sometimes called "rvalues" because in an assignment expression they may only appear on the right side.

simd_ lvalue expressions are produced and required according to rules analogous to those for scalar expressions. The description of each operator accepting or producing a simd_ expression notes when an operator must be an lvalue and when the result is an lvalue. When no such comment is present, the operands need not be lvalues and the result is not an lvalue.

5.4.5 Storage

The exact amount of storage per virtual PE used for data elements of a simd_ object is implementation-dependent, subject to the following constraints. The individual values of any particular simd_ object must be stored identically in each virtual PE. If a length argument was used in the simd_ object's declaration, at least that many bits of each virtual PE's memory must be used to store the object, though more may be used. All simd_ objects declared with a particular length must be stored in the same number of bits of virtual PE memory.

The C operator sizeof does not give information about the amount of PE storage used to store simd_ objects. Rather, it reports the size of the implementation-dependent host object used to store information about the simd_ object. The bits() method returns as an unsigned value the declared length of any simd_ object. The declared length of a boolean simd_ object is 1. The declared length of a floating point simd_ object is implementation dependent. The following fragment prints 16.

```
simd_int a(16);
printf("%u", a.bits());
```

There is no provision for finding the number of bits actually used to store a `simd_` object, which is greater than or equal to the number returned by `bits()`.

5.4.6 Conversions

The type conversion rules governing ordinary C types are extended in Porta-SIMD to incorporate the `simd_` types as naturally as possible. This section's subsections describe each new or extended conversion rule in Porta-SIMD. The first subsection treats the actual representation changes involved in each type conversion. The rest define which conversions are performed under which circumstances.

Every conversion is performed entirely within a single virtual machine.

5.4.6.1 Representation Changes

The basic rules governing representation changes for scalar integer types apply to conversions among `simd_` integer types as well. These representation changes are performed in all PEs of the virtual machine on which the parallel value resides, regardless of whether the PE is enabled or not. (Sections 5.4.7.1 and 5.4.8 discuss the enable status of PEs.)

The most basic conversion is between objects of the same type but different lengths. An unsigned `simd_` object is converted to a shorter unsigned `simd_` object by discarding the excess high-order bits. It is converted to a longer unsigned `simd_` object by filling the additional high-order bits with zeros. If signed integer `simd_` objects are represented in two's-complement form (as in all current implementations), the rules are very similar. On conversion to a shorter object, excess high-order bits are discarded. When converting to a longer object, the additional high-order bits are filled with copies of the sign bit.

Conversion between a base `simd_` type and one of its derived types, or between two derived types of the same base types, requires no change of representation. However, conversion is allowed only to derived types with an architecture identifier which is a subset of the architecture identifier of the virtual machine on which the data resides.

Conversion between signed and unsigned integer `simd_` types always begins by converting the original data, still in its own type, to the same length as the destination type. If two's-complement form is used for signed integers, no further conversion is needed between integer `simd_` types.

A non-boolean `simd_` type is converted to a boolean `simd_` type very simply. Every non-zero element takes the boolean value one (true), and every zero element takes the value zero (false). A boolean `simd_` type is converted to an integer `simd_` type by treating it as a very short unsigned object; it is simply zero-extended to the desired length. These rules are consistent with C's treatment of boolean values (e.g., the test expressions of `if`, `for`, and `while` statements, and the value of relational operators such as <).

Conversion between floating point and integer `simd_` types is done exactly like it is done for scalar types. Conversion between floating point and boolean `simd_` types is done in two steps, converting first to an unsigned `simd_` type, and from that to the final type.

A scalar value can be converted to a `simd_` value of the same type by replication; every element of the resulting `simd_` object has the scalar's value. Conversion of a scalar value to a `simd_` value of a different basic type proceeds in two steps: conversion to a `simd_` value of the scalar's type, followed by conversion to the desired `simd_` type. A `simd_` value cannot be converted to a scalar.

5.4.6.2 Casting

C++ allows all legal conversions to be invoked explicitly using a type cast. Due to limitations inherent in the implementation technology, Porta-SIMD does not provide this much flexibility. Porta-SIMD allows casts of all basic `simd_` types to unsigned `simd_` types. However, it does not allow the opposite casts, from unsigned to other basic `simd_` types. It also does not allow casts between other basic `simd_` types. (E.g., a cast from `simd_bool` to `simd_int` is not allowed.) A scalar value cannot be cast to a `simd_` type.

Any conversion not possible with a cast can be performed with a simple assignment statement (section 5.4.6.3). Here are some acceptable casts.

```
simd_unsigned     su;
simd_G_unsigned  sGu;
simd_int          si;
simd_2aC_int   s2aCi;
simd_bool         sb;
simd_float        sf;

(simd_unsigned)   si;
(simd_unsigned)   saCi;
(simd_unsigned)   sb;
(simd_unsigned)   sGu;
(simd_unsigned)   sf;
(simd_G_unsigned) s2aCi;
(simd_G_unsigned) sb;
(simd_G_unsigned) su;
```

Here are some casts that are *not* allowed.

```
(simd_int)      su;
(simd_int)      sb;
(simd_bool)     su;
(simd_bool)     si;
(simd_float)    su;
(simd_unsigned) 1;
(simd_int)      1;
(simd_bool)     1;
```

5.4.6.3 Assignment

All legal conversions can be accomplished through the simple assignment statement. (The simple assignment statement uses the = operator.) Simply write an object of the desired type on the left side of the =, and the value to be converted on the right side.

5.4.6.4 Usual Unary Conversions

C and C++ define certain "usual conversions" that are performed implicitly during expression evaluation. Their purpose is to convert all the operands of an operator to a common type before performing the operation. Porta-SIMD also converts the operands to a common basic `simd_` type and common length before performing an operation. However, the usual conversions also ensure that the common type will be one of a very small set of types. Porta-SIMD extends this set of types to include all the `simd_` types. As a result, there is no implicit conversion of `simd_` types during evaluation of unary operators.

5.4.6.5 Usual Binary Conversions

The following rules are added to the set of usual binary conversions, and are applied, in order, whenever an operator has one or more parallel operands.

1. If one operand is scalar and the other is parallel, convert the scalar operand to its corresponding `simd_` type.

2. If one of the operands is integer or boolean and the other is floating point, convert the integer or boolean to the floating point type.

3. If the operands have different lengths, lengthen the shorter to the length of the longer. If one of the operands is boolean, this implicitly converts it to an unsigned integer.

4. If one operand is signed and the other is unsigned, convert the signed operand to its unsigned equivalent. (This is what C and C++ do with scalars.)

5. If the operand types have different architecture identifiers, convert both to the same type. This may be their common base type, or any derived `simd_` type which has an architecture identifier specifying only architectural features present in both operand types. The exact type selected is implementation dependent.

These ensure that the operands have the same base type, length, and architecture identifier.

Of course, while a Porta-SIMD implementation is required to evaluate expressions as if these rules had been used, it is not required to literally perform the specified conversions if the same result can be achieved by a more efficient method.

5.4.6.6 Function Arguments and Return Values

If an expression appearing as an argument in a function call does not match the type of that argument as declared in the function declaration, it is cast to the declared type of the argument. Similarly, if the expression appearing in a `return` statement does not match the return type in the function declaration, it is cast to the declared return value type. In both cases, it is an error if the resulting cast is not allowed by Porta-SIMD.

5.4.7 Expressions

`simd_` values may be accessed only by mechanisms explicitly provided by Porta-SIMD. With a few exceptions discussed in this section, all C operators may be applied to `simd_` objects. The use of each operator with `simd_` types is discussed in detail below, including the precise operand types allowed, whether to apply the usual binary conversions, and the type of the result.

All the C operators implemented by Porta-SIMD perform the same computation as in C and C++, but do it in every enabled PE when the operands are parallel. They have the same precedence and associativity as in C and C++.

Except as explicitly described in sections 5.4.7.4, 5.4.12.5 and 5.4.17.3.2, all `simd_` objects within a single expression must reside on the same virtual machine. Some violations of this rule cannot be detected at compile time. Depending on the implementation, such violations may cause the program's execution to be aborted with a message, or may be silently ignored. In any case, expressions violating this rule have undefined results.

5.4.7.1 Enabled and Disabled PEs

Operations and methods that modify `simd_` objects (including temporary objects created implicitly during expression evaluation) are performed only in enabled PEs. However, conversions are performed in all PEs, enabled or not. There are a few additional operations that modify disabled PEs; this is explicitly stated where each such operation is described. Any operation which does not have such a statement in its description operates only in enabled PEs. Section 5.4.8 defines when PEs are enabled.

5.4.7.2 Primary and Postfix Operators

An array of `simd_` objects may be declared. Because it is a host array, such an array of `simd_` objects may be subscripted with `[]` like any other host array.

There are no parallel function calls, so the `()` (function call) operator cannot be applied to `simd_` objects. Of course, scalar functions can return `simd_` objects.

There are also no parallel structures, so the `.` (direct selection) and `->` (indirect selection) operators cannot be applied to `simd_` values. Scalar structures may contain `simd_` objects, though. The entire `simd_` object, represented by its host-resident "handle" or "descriptor", is contained in the structure.

The postfix operators `++` (post-increment) and `--` (post-decrement) may be applied to numeric `simd_` objects only. Unfortunately, Porta-SIMD is limited by its implementation within C++ here, and is unable to provide the proper access-then-operate semantics. Therefore, these postfix operators are currently exactly equivalent to their prefix forms. A future implementation of Porta-SIMD may correctly implement the postfix forms of these operators. Until then, programs should avoid these postfix operators and use only their prefix forms.

5.4.7.3 Unary Operators

The prefix `++` (pre-increment) and `--` (pre-decrement) operators and unary `-` (negate) operator may be applied to numeric `simd_` type lvalues, and produce an lvalue of the same type.

The `~` (bitwise not) operator may be applied to any integer or boolean `simd_` type, producing a value of the same type.

The `!` (logical not) operator may be applied to any `simd_` type, and produces a boolean `simd_` value with an implementation dependent architecture identifier.

The `sizeof` and cast operators were discussed in sections 5.4.5 and 5.4.6.2, respectively.

The `&` (address of) operator may be applied to a `simd_` object, and produces a scalar pointer to the object. (The pointer points to the object's host "handle".) The unary `*` (indirection) operator may be applied to a pointer to a `simd_` object (initialized with the `&` operator); it produces a `simd_` object which may be used like any other `simd_` object. There are no scalar pointers to individual elements of `simd_` objects. Section 5.4.17.2 describes `simd_` pointers.

5.4.7.4 Assignment Operator

As mentioned in section 5.4.6.3, the assignment operator (`=`) has two arguments and stores into the left argument the value of the right argument. The left argument must be an lvalue. If the right argument is a `simd_` value, the left argument must be, too. The arguments may have any combination of types, provided a conversion is defined in section 5.4.6 from the type of the right argument to the type of the left argument.

If both arguments are `simd_` values, then they must be on the same virtual machine, or else on virtual machines which were declared with identical argument values and identical labeling in their architecture identifiers. This case provides trivial communication between distinct virtual machines having identical size and shape.

5.4.7.5 Arithmetic Operators

The binary arithmetic operators include `*` (multiplication), `/` (division), `%` (remainder), `+` (addition), and `-` (subtraction). These operators may be applied to any combination of scalar and parallel numeric operands. The single exception is that the operands of `%` must be integers. The usual binary conversions apply to the operands, and the result is the type to which the operands were converted.

The corresponding binary arithmetic assignment operators are `*=`, `/=`, `%=`, `+=`, and `-=`. These may be applied to any combination of scalar and parallel numeric operands, provided the left operand is a parallel lvalue. Again, the operands of `%=` must be integers. The usual binary conversions apply to the operands, and the result is converted to the type of the left operand and stored in it. The result is the modified left operand, an lvalue.

Boolean `simd_` types may appear as operands to these operators under the following conditions. Only one operand may be boolean, and the other operand must have a parallel numeric type. A boolean object may not be the left operand of an arithmetic assignment operator. The conversion rules are not affected by the presence of a boolean operand.

5.4.7.6 Shift Operators

The shift operators include `<<` (left shift) and `>>` (right shift). These operators may be applied to any combination of scalar and parallel integer operands. The usual unary conversions apply to each operand, but the usual binary conversions do not apply. However, if one operand is scalar and the other is parallel, the scalar operand is converted to a `simd_` operand of its own basic type and the same architecture identifier as the parallel operand. The operands are also converted to a common architecture identifier, using the same rule used by the usual binary conversions. The result has the type, after all conversions, of the left operand.

The corresponding shift assignment operators are `<<=` and `>>=`. These operators may be applied to any combination of scalar and parallel integer operands, provided the left operand is a parallel lvalue. The same conversion rules apply as for the non-assignment forms. The result is stored in the left operand. The result is the modified left operand, an lvalue.

Boolean `simd_` types may appear as operands to these operators under the following conditions. Only one operand may be boolean, and the other operand must have a parallel integer type. A boolean object may not be the left operand of a shift assignment operator. The conversion rules are the same as in the absence of a boolean operand, except that a boolean left operand is converted to an unsigned `simd_` type the same length as the right operand.

5.4.7.7 Bitwise Operators

The bitwise operators include `&` (and), `^` (xor), and `|` (or). These operators may be applied to any combination of scalar and parallel integer operands. The usual binary conversions apply to the operands, and the result is the same type that the operands are converted to.

The corresponding bitwise assignment operators are `&=`, `^=`, and `|=`. These may be applied to any combination of scalar and parallel integer operands, provided the left operand is a parallel lvalue. The usual binary conversions apply to the operands, and the result is converted to the type of the left operand and stored in it. The result is the modified left operand, an lvalue.

Boolean `simd_` types may appear as operands to these operators under the following conditions. Only one operand may be boolean, and the other operand must have a parallel integer type. A boolean object may not be the left operand of a bitwise assignment operator. The conversion rules are not affected by the presence of a boolean operand.

5.4.7.8 Relational Operators

The relational operators include < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal), and != (not equal). These operators may be applied to any combination of numeric scalar, numeric parallel, and boolean parallel operands. The usual binary conversions apply to the operands. The result is boolean parallel, with the same architecture identifier as the type to which the operands were converted.

5.4.7.9 Logical Operators

The logical operators include `&&` (logical and), and `||` (logical or). These operators may be applied to any combination of scalar and parallel numeric operands and parallel boolean operands. The usual unary conversions apply, but not the usual binary conversions. If one operand is scalar and the other is parallel, the scalar operand is converted to a `simd_` operand of its own basic type and the same architecture identifier as the parallel operand. The operands are also converted to a common architecture identifier, using the same rule used by the usual binary conversions. The result is boolean, with the same architecture identifier as the converted operands.

5.4.7.10 Other C Operators

The ternary `?:` (conditional) operator may not have a `simd_` first operand. The second and third operands may be `simd_` expressions, provided the expressions evaluate to exactly the same type. (Note that this identical-type requirement implies that either both or neither of the second and third operands have a `simd_` type.)

The comma operator is the same in Porta-SIMD as in C and C++. It may separate any pair of expressions.

5.4.8 Flow Control

Porta-SIMD provides a parallel conditional `IF` statement similar to the scalar `if` statement. The syntax is

```
IF ( <simd expr> )
  <statements>
ELSE
  <statements>
ENDIF
```

`<simd expr>` represents any expression which evaluates to a value of a `simd_` type. `<statements>` represents any statements, including nested `IF` statements properly paired with their nearest following `ENDIF`s. The `ELSE` and its following `<statements>` may be left out.

The `<simd expr>` is evaluated once, and effectively assigned into a `simd_bool` temporary variable. During execution of the statements between `IF` and `ELSE` (or `ENDIF` if there is no `ELSE`), only PEs where the temporary variable is true are enabled. During execution of the statements between the ELSE and ENDIF, the only enabled PEs are those where the temporary variable is false. When a nested IF statement is executed, PEs already disabled before the nested IF statement was encountered remain disabled throughout its execution. Comments in the following example show which PEs are enabled, by giving the expression which must be true in every enabled PE.

```
simd_bool a, b;
/* Initialize a and b. */
IF (a)
  /* a */
  IF (b)
    /* a && b */
  ELSE
    /* a && !b */
  ENDIF
  /* a */
ELSE
  /* !a */
  IF (b)
    /* !a && b */
  ENDIF
  /* !a */
ENDIF
```

It is important to recognize that every statement between IF and ENDIF is always executed, though only in the appropriate PEs. This includes scalar expressions, which are executed in the host as usual. The following code fragment illustrates the potentially surprising results.

```
simd_bool a;
/* Initialize a. */
int x = 0;
IF (a)
  x++;                /* bad style */
ELSE
  x++;                /* bad style */
ENDIF
printf("x=%d", x);
```

It prints x=2. It is usually wise not to modify scalar variables within a parallel IF statement. An exception is a scalar variable local to only one part of the IF statement (the statements following either IF or ELSE, but not both).

Some SIMD languages define a parallel if statement which does not execute <statements> unless at least one PE is enabled. Such an if statement cannot be implemented without the optional architectural feature "goR", so it cannot be standard in an optimally portable SIMD language. However, section 5.4.18.4 describes how to express such a statement in Porta-SIMD when the target architecture supports it. That section also describes Porta-SIMD's parallel WHILE statement.

A parallel ALL statement can be used to temporarily enable all PEs. The syntax is

```
ALL
  <statements>
ENDALL
```

It enables all PEs for execution of `<statements>`, then restores the enable status in use before `ALL` was executed. It thereby ensures that all PEs execute `<statements>`, even PEs currently disabled by `IF` or `WHILE` statements.

5.4.9 Special Operations

The method `pe_number()` may be applied to any integer `simd_` lvalue, and has no arguments. It stores into the lvalue in each PE the unique number of that (virtual) PE. The PE number is treated as an infinite precision unsigned `simd_` value, the high order bits of which are truncated on assignment to the lvalue. The result is the modified lvalue.

```
simd_unsigned a;
a.pe_number();
```

The method `set()` may be applied to any `simd_` lvalue. It has two arguments: (1) an unsigned scalar PE number (`unsigned`), (2) a scalar value convertible to the basic type of the `simd_` lvalue. The second argument's value is stored into the lvalue in the PE identified by the first argument. There is no result.

```
simd_unsigned a;
unsigned pe = 5, value = 25;
a.set(pe, value);
```

The method `bits()` is described in section 5.4.5. It may be applied to any `simd_` object. It has no arguments, and returns a scalar unsigned value which is the declared number of bits in the `simd_` object.

The method `m_ARCHID()` may be applied to any `simd_` object. (The underscore is omitted if the null architecture identifier is used.) It has no arguments, and returns a pointer to the virtual machine on which the `simd_` object is stored. The architecture identifier of the returned pointer is that specified in the method name. If the architecture identifier specified is not a subset of the architecture of the `simd_` object to which the method is applied, a null pointer is returned.

```
simd_2aCG_mach mach;
simd_2aCG_unsigned u(mach);
simd_mach *m_ptr;
m_ptr = u.m();
m_ptr = u.m_2aC();
m_ptr = u.m_gC();  // Returns NULL
```

The method `pes()` may be applied to any virtual machine object or `simd_` object. It has no arguments. It returns an unsigned scalar value, which is the number of PEs in the virtual machine, or in the virtual machine on which the `simd_` object is stored.

```
simd_mach m;
unsigned num_pes;
num_pes = m.pes();
simd_bool b(&m);
num_pes = b.pes();
```

5.4.10 Providing Architectural Features

The remaining sections of this language definition describe language features which provide access to optional architectural features. The sections are named for the features they discuss. The methods described in each remaining section may only be applied to `simd_` objects with an architecture identifier which subsumes the architectural feature provided.

5.4.11 Labeling

5.4.11.1 1, 2, ..., N (Cartesian Coordinates)

The method `pe_coord()` may be applied to any integer `simd_` lvalue whose architecture identifier contains a cartesian labeling. It takes one argument: a scalar unsigned dimension number which must be less than the number of dimensions. Dimensions are numbered from zero. The PE's coordinate in the specified dimension is stored in the lvalue. The result is the modified lvalue.

Whenever a file's architecture identifier contains or subsumes a cartesian labeling "L", the type `label_L` is defined. For example, `label_1` and `label_2` are defined in files which have a 2-D cartesian labeling as part of their architecture identifier. The value of a `label_` object specifies a particular PE in a particular virtual machine. `label_` objects are useful for converting between a PE's coordinates and PE number.

A `label_` object is always initialized as it is declared, using one of the type's two forms of declaration. All declarations take a pointer to a virtual machine as the first argument. The pointer type must contain the same labeling as the `label_` object being declared. The first form of declaration has two arguments, the second of which is the PE number of a PE on the virtual machine pointed to by the first argument. This form is the same for all `label_` types. The second form has $n + 1$ unsigned scalar arguments numbered from the left (i.e., the first argument is number 1), where argument i is the coordinate of the specified PE in dimension $i - 1$ and n is the number following `label_` in the type name of the `label_` object being declared. The second form of declaration for type `label_N` is handled differently from the fixed-dimension types just discussed. The second argument in a `label_N` declaration is a pointer to an array of PE coordinates; the array must have the same number of elements as the virtual machine pointed to by the first argument has dimensions, and each element i is the coordinate in dimension i.

The following methods may be applied to `label_` objects. Method `dimens()` takes no arguments and returns the number of dimensions in the virtual machine on which the PE specified by the object exists; `mach()` takes no arguments and returns a pointer to that virtual machine. Method `coord()` takes an unsigned scalar argument and returns the PE's coordinate in that dimension. Method `pe_number()` invoked without arguments returns the PE number of the PE identified by the object; invoked with an unsigned scalar argument it sets the object to identify the PE with that number (on the same virtual machine the object already specifies) and returns the argument value. The array index operator (`[]`) may be applied to `label_` objects. The braces must enclose an unsigned scalar dimension number, and the value of the expression is the PE's coordinate in that dimension; if this expression appears as the left side of a simple assignment operator (=), the coordinate is set to the assignment operator's right argument and the new value is then returned. If an illegal PE number or coordinate value is specified as an argument to any of these methods or operators, the result is undefined.

5.4.11.1.1 2 (2-D Cartesian Coordinate) Constants are defined which name the two dimensions. They are `dimen_x` and `dimen_y`, and they have values 0 and 1 respectively.

Several methods may be applied only to `simd_` lvalues with a 2-D cartesian labeling. The `bilinear()` method computes and stores $ax + by + c$ in the lvalue to which it is applied. The coefficients a, b, and c are floating point scalar arguments, and the floating-point result is truncated before being stored. x and y represent each PE's coordinates in the zero and one dimensions, respectively. `bilinear()` may be applied only to integer `simd_` lvalues. This code fragment computes $x + .5y - 3$.

```
    simd_2_int a;
    a.bilinear(1, .5, -3);
```

The biquadratic() method differs from bilinear() only in its arguments and the value it computes. biquadratic() has six floating point scalar arguments named a through f, respectively. It computes and stores in the lvalue to which it is applied the value of $ax^2 + by^2 + cxy + dx + ey + f$.

5.4.11.2 p2 (Level and 2-D Coordinate)

The pe_coord() method defined for cartesian coordinate labelings may also be applied to integer simd_ lvalues with a p2 labeling. Each PE's coordinate within its level is stored into the lvalue. The pe_level() method may be applied to any integer simd_ lvalue with a p2 labeling. It has no arguments, and stores in the lvalue in each PE that PE's level number. The result is the modified lvalue.

5.4.12 Communication (C)

Architecture features for communication, and especially adjacent communication, have more interactions than any other set of features. This causes a profusion of variations on the basic communications operations: send() and fetch().

One kind of variation is the way in which the direction of communication is specified. Several types, with names beginning direction_, are used to specify communication patterns. The remainder of each such type name is a feature name. The direction_ types are used as arguments to send() and fetch() operations. The architecture identifier of the simd_ object to which these operations are applied must support the feature named by the direction_ type which is used to specify the communication pattern. In addition, each direction_ type may only be used in a source file whose target architecture subsumes the feature named by the direction_ type. The direction_ types are described in the subsections which follow.

The send() method may be applied to any simd_ value with a communication feature. The fetch() method may be applied to any simd_ lvalue with a communication feature. Both methods have two arguments: (1) a simd_ value, (2) a direction_ type value. For send() only, the first argument must be an lvalue. The first argument must be on the same virtual machine as the object to which send() or fetch() is applied, or else the two virtual machines must have been declared with exactly the same arguments and have the same labeling in their architecture identifiers. send() causes each enabled PE to send the value to which send() is applied to the PE specified by the direction_ argument, where the value is then stored into the first argument. The value is stored regardless of the enable status of the receiving PE. fetch() causes each enabled PE to fetch the value of the first argument from the PE specified by the direction_ argument, and store the fetched value into the simd_ lvalue to which fetch() is being applied. The value is fetched and stored by each enabled PE, regardless of the enable status of the PE from which the value is fetched. The result of fetch() is the modified lvalue to which it is applied. No result is returned by send(). If the basic types of the simd_ value to which send() or fetch() is applied and the simd_ first argument are not the same, the necessary conversion is performed before the communicated data is stored.

5.4.12.1 aC (Adjacent)

The type direction_aC describes ordinary "aC" adjacent communication. Its declaration requires two arguments: (1) unsigned scalar dimension number, (2) signed integer scalar direction along the dimension. The method dimen() with no arguments returns the dimension number of the `direction_aC` value it is applied to. With one unsigned scalar argument, it sets the dimension number to the argument value and returns the modified dimension number. The method `dir()` with no arguments returns the direction of the `direction_aC` value it is applied to. With one signed integer scalar argument, it sets the direction to the argument value and returns the modified direction. The direction value must always be one of: -1, 0, 1. Each PE initiating communication does so with the PE whose label differs by `dir()` in dimension `dimen()`. A `direction_aC` value may be assigned to a `direction_aC` lvalue.

```
simd_2aCG_mach m;
simd_2aC_unsigned u_aC(m);
simd_unsigned u(m);
direction_aC east(0,1);
u_aC.send(u, east);
u = u_aC.fetch(u, east);
```

5.4.12.1.1 d (Diagonal) The type `direction_adC` describes "adC" adjacent diagonal communication. Its declaration requires one argument: (1) unsigned scalar number of dimensions. The method `dimens()` with no arguments returns the number of dimensions of the `direction_adC` value it is applied to. The operator `[]` with one unsigned argument, a dimension number, produces an lvalue which is the `direction_adC`'s direction along that dimension. Only the values 1, 0, and -1 may be assigned to this lvalue. The direction along each dimension must be defined before a `direction_adC` value is used as an argument to a `send()` or `fetch()` method. Each PE initiating communication does so with the PE whose label differs by `[i]` along each dimension `i`. A `direction_adC` object may be initialized by declaring it with a single argument of type `direction_aC`. Values of type `direction_aC` and `direction_adC` may be assigned to `direction_adC` lvalues.

```
simd_2adCG_mach m;
simd_2adC_unsigned u_adC(m);
simd_unsigned u(m);
direction_adC northwest(2);
northwest[0] = -1;
northwest[1] = 1;
u_adC.send(u, northwest);
u = u_adC.fetch(u, northwest);
```

5.4.12.1.2 l (Local) The type `direction_alC` describes "alC" adjacent communication with local choice of direction. Its declaration requires two arguments: (1) unsigned scalar dimension number, (2) signed integer `simd_` direction along the dimension. The method `dimen()` with no arguments returns the dimension number of the `direction_alC` value it is applied to. With one unsigned scalar argument, it sets the dimension number to the argument value and returns the modified dimension number. The method `dir()` with no arguments returns the direction in each PE of the `direction_alC` value it is applied to. With one signed integer `simd_` argument, it sets the direction to the argument value in each PE and returns the modified direction. The direction value in each PE must always be one of: -1, 0, 1. The `simd_` direction value must be on the same virtual machine as the `simd_` value to which `send()` or `fetch()` is applied. Each PE initiating communication does so with the PE whose label differs by `dir()` in dimension `dimen()`. A `direction_alC` object may be initialized by declaring it with a single argument of type `direction_aC`. Values of type `direction_aC` or `direction_alC` may be assigned to a `direction_alC` lvalue.

```
simd_2alCG_mach m;
simd_2alC_unsigned u_alC(m);
simd_unsigned u(m);
simd_int i(m);
/* ... */
direction_alC east(0,i);
u_alC.send(u, east);
u = u_alC.fetch(u, east);
```

5.4.12.1.3 d (Diagonal) and l (Local) The type direction_adlC describes "adlC" adjacent diagonal communication with local choice of direction. Its declaration requires one argument: (1) unsigned scalar number of dimensions. The method dimens() with no arguments returns the number of dimensions of the direction_adlC value it is applied to. The operator [] with one unsigned argument, a dimension number, produces a simd_ lvalue which is the direction_adlC's direction along that dimension in each PE. Only the values 1, 0, and -1 may be assigned to this lvalue in each PE. The direction along each dimension must be defined before a direction_adlC value is used as an argument to a send() or fetch() method. The simd_ direction value in every dimension must be on the same virtual machine as the simd_ value to which send() or fetch() is applied. Each PE initiating communication does so with the PE whose label differs by its value of [i] along each dimension i. A direction_adlC object may be initialized by declaring it with a single argument of type direction_aC, direction_adC, or direction_alC. Values of type direction_aC, direction_adC and direction_alC may be assigned to direction_adlC lvalues.

```
simd_2adlCG_mach m;
simd_2adlC_unsigned u_adlC(m);
simd_unsigned u(m);
simd_int i(m), j(m);
/* ... */
direction_adlC some_dir(2);
some_dir[0] = i;
some_dir[1] = j;
u_adlC.send(u, some_dir);
u = u_adlC.fetch(u, some_dir);
```

5.4.12.1.4 w (Wrap) Wrap-around communication provided by the "w" option to "aC" is expressed in Porta-SIMD as a variation on the send() and fetch() method names, rather than on their argument types. The characters _wrap are added to send and fetch to specify that computation of the remote PE label is done using modulo arithmetic. (This is true independent of other communication variations, including "aC"s "d" and "l" modifiers. Like other optional features, the presence of support for wrap-around communication does not prevent the use of methods which do not require it. Therefore, both _wrap and regular communication methods may be used on simd_ objects which support wrap-around communication.)

```
simd_2adwlCG_mach m;
simd_2awC_unsigned u_awC(m);
simd_unsigned u(m);
direction_aC east(0,1);
u_awC.send_wrap(u, east);
u = u_awC.fetch_wrap(u, east);
```

```
simd_unsigned u_2adwlC(m);
simd_int i(m), j(m);
/* ... */
direction_adlC some_dir(2);
some_dir[0] = i;
some_dir[1] = j;
u_adwlC.send_wrap(u, some_dir);
u = u_adwlC.fetch_wrap(u, some_dir);
```

5.4.12.2 mC (Pyramid)

All the adjacent communication methods for variations of the "aC" feature may also be applied to any `simd_` object which has "mC" in its architecture identifier. This provides communication within levels of the pyramid. Communication between levels is supported by additional methods.

The type `direction_mC` is scalar and enumerated, with these values: `ne`, `se`, `sw`, `nw`. The methods `send_child()` and `fetch_child()` have the same arguments and same results as the basic `send()` and `fetch()` methods. They both have a `direction_mC` type second argument which specifies which child each PE communicates with. PEs without children are like PEs in adjacent communication when no wrap is used: sent data is discarded and fetched data is zero.

The `fetch_parent_replicate()` method has only one argument, the normal `fetch()` method's first argument, and the same result as the normal `fetch()` method. Each PE fetches data from its parent PE. The single PE without a parent fetches zeros.

The `send_parent_with_COMBINE()` methods are named by replacing `COMBINE` with one of: `overwrite`, `and`, `or`, `xor`, `add`, `max`, `min`, `mul`. These methods have one argument, the the normal `send()` method's first argument. Each PE sends data to its parent, and the parent combines the values it receives using the specified combining operation. The combining operations are described in section 5.4.13. Data sent by the single PE without a parent is discarded.

5.4.12.3 cC (Cube Connected Cycles)

The type `direction_cC` is scalar and enumerated, with these values: `s`, `p`, `l`. The `direction_cC` type second argument is used with the `send()` and `fetch()` methods to do CCC communication.

5.4.12.4 pC (Preselected Permutation)

The type `direction_pC` is a scalar type which identifies one of a set precomputed permutations. I do not have information on the way such permutations are specified in the GF11, which is the only machine I know of to use the "pC" feature. Therefore, I have not defined how values of type `direction_pC` are specified. However, once a `direction_pC` value is available, it is used as the second argument to the `send()` and `fetch()` methods to do "pC" communication.

5.4.12.5 gC (Global)

The type `direction_gC` is simply an unsigned integer `simd_` type. The value of a `direction_gC` in each PE is used as a PE number, identifying the PE with which to communicate. A `direction_gC` value is used as the second argument to the `send()` and `fetch()`

methods to do "gC" communication. The `direction_gC` value must be on the same virtual machine as the `simd_` value to which the `send()` or `fetch()` operation is applied. Unlike all other communication operations, `send()` and `fetch()` operations with a `direction_gC` type second argument may communicate between any two virtual machines. The first argument and the object to which the operation is applied may be on any two virtual machines. If the second argument in any PE names a nonexistent PE in the virtual machine being communicated with, the result of communication is undefined in all PEs.

```
simd_gC_mach m;
simd_gC_unsigned u_gC(m);
simd_unsigned u;
simd_unsigned v(m);
direction_gC d(m);
/* ... */
u_gC.send(u, d);
v = u_gC.fetch(u, d);
```

5.4.13 Collision Resolution, Write (W)

Collision resolution on write is specified by appending a string to the name of the `send()` method to indicate the combining operation to use. The string appended has the form `_with_OP`, where `OP` is one of the combining operators described in the remainder of this section. This string is appended after `_wrap`, if `_wrap` is also used. Some examples are shown after all the combining operators are described.

5.4.13.1 sW (Select)

The combining operator `overwrite` may be used when the `simd_` object to which `send()` is applied supports the "sW" feature. `overwrite` specifies that a PE receiving multiple values store one (any arbitrary one) and discard all others.

5.4.13.2 lW (Logically Combine)

The combining operators `and`, `or`, and `xor` may be used when the `simd_` object to which `send()` is applied supports the "lW" feature. These combining operations specify that a PE receiving multiple values combine them with the named bitwise operation and store the result.

5.4.13.3 aW (Add or Compare)

The combining operators `add`, `max`, and `min` may be used when the `simd_` object to which `send()` is applied supports the "aW" feature. These combining operations specify that a PE receiving multiple values combine them with the named arithmetic operation and store the result. `add` means combine by adding. `max` means combine by selecting the maximum value. `min` means combine by selecting the minimum value.

5.4.13.4 mW (Multiply)

The combining operator `mul` may be used when the `simd_` object to which `send()` is applied supports the "mW" feature. This combining operation specifies that a PE receiving multiple values multiply them together and store the result.

Here are some examples of collision resolution on write.

```
simd_gCmW_mach m_gC;
simd_2awlCmW_mach m_2aC;
simd_gCmW_unsigned u_gC(m_gC);
simd_2awlCmW_unsigned u_2aC(m_2aC);
direction_gC dg(m_gC);
simd_unsigned u(m_2aC);
/* ... */
direction_alC da(2, u);
u_gC.send_with_add(u, dg);
u_2aC.send_with_mul(u, da);
u_2aC.send_wrap_with_xor(u, da);
```

5.4.14 Collision Resolution, Fetch (F)

Collision resolution on fetch is specified by appending a string to the name of the `fetch()` method to indicate the combining operation to use. The string appended has the form `_with_OP`, where `OP` is one of the combining operators described in the remainder of this section. This string is appended after `_wrap`, if `_wrap` is also used. Some examples are shown after all the combining operators are described.

5.4.14.1 sF (Select)

The combining operator `select` may be used when the `simd_` object to which `fetch()` is applied supports the "sF" feature. `select` specifies that a PE receiving fetch requests for its value from multiple PEs selects one arbitrarily and sends its value to the selected fetching PE, ignoring all the other requests. This means exactly one PE fetching from any particular single PE is successful. Unsuccessful PEs fetch no data at all.

5.4.14.2 rF (Replicate)

The combining operator `replicate` may be used when the `simd_` object to which `fetch()` is applied supports the "rF" feature. `replicate` specifies that all PEs fetching from any particular PE are successful. The data of any PE receiving multiple fetch requests is replicated to satisfy all the requests.

Here are some examples of collision resolution on fetch.

```
simd_gCmW_mach m_gC;
simd_2awlCmW_mach m_2aC;
simd_gCmW_unsigned u_gC(m_gC);
simd_2awlCmW_unsigned u_2aC(m_2aC);
direction_gC dg(m_gC);
simd_unsigned u(m_2aC);
/* ... */
direction_alC da(2, u);
u = u_gC.fetch_with_replicate(u, dg);
u = u_2aC.fetch_with_select(u, da);
u = u_2aC.fetch_wrap_with_replicate(u, da);
```

5.4.15 Piped Communication (P)

Piped communication is specified by appending the string `_across_disabled` to the name of the `send()` or `fetch()` method, and also adding a third argument. The string _across_disabled is appended after _wrap, if _wrap is also used. The third argument is an unsigned scalar distance value. This distance must not exceed the number of PEs the virtual machine doing the communicating has in any dimension involved in the communication. For non-diagonal communication, this is the dimension specified by direction_aC. For diagonal communication, this is any dimension with a non-zero value in direction_adC. As specified in the taxonomy, there may not be any enabled PEs in the direct path of communicating PEs.

Some examples are shown after the "u" and "c" modifiers of "P" are described.

5.4.15.1 u (Unlimited)

Unlimited piped communication is specified by adding the third argument to send() and fetch() methods to specify the distance, as in regular piped communication, but without appending any string to the names of these methods. The restriction concerning enabled PEs between communicating PEs is removed.

5.4.15.2 c (Copy)

Copy piped communication is specified just like regular piped communication, except the string _copy_across_disabled is used instead of _across_disabled. As specified in the taxonomy, every PE directly between communicating PEs receives a copy of the data communicated, and all of these PEs must be disabled. The intermediate copies of the data are stored in the same simd_ object as the regular communicated data.

Here are some examples of piped communication.

```
simd_2awCucP_mach m;
simd_2awCucP_unsigned u(m);
direction_aC d(0, 1);
simd_unsigned v(m);
unsigned dist;
/* ... */
u.send_wrap_across_disabled(v, d, dist);
v = u.fetch(v, d, dist);
v = u.fetch_copy_across_disabled(v, d, dist);
```

5.4.16 Cut-Through Communication (T)

Cut-through communication is specified by replacing the second argument of the send() method with a value of a switch_ type. There is no fetch() for cut-through communication. The switch_ type specifies the programming of each PE's connection to the communication network; it must be programmed, as described in succeeding paragraphs, before it is used. There are sixteen switch_ types, the result of four independent boolean modifiers. Each modifier may be used only when it is supported by the architecture identifier of the simd_ object to which send() is applied with the switch as an argument. The modifiers are as follows.

d Corresponds to the "d" modifier of the "aC" feature.

c Corresponds to the "c" modifier of the "T" feature.

r Corresponds to the "r" modifier of the "T" feature. "r" must be used when "T" has the "r" modifier, and may be used when "T" does not. This is the opposite situation of the other modifiers, which may be used when there is architectural support, and must not be used when there is not. This is because "T" 's "r" limits the architecture, unlike the other features which extend the architecture.

l Corresponds to the "l" modifier of the "T" feature.

The `switch_` types are named by appending the selected modifier letters, in the order just listed. The underscore is omitted if no modifier letters are selected.

A `switch_` object represents a switch at every PE. Switches have ports and shorts. A port is a "wire" to an adjacent PE, or possibly to the switch's PE. A short is a logical connection between two or more ports. "Short" is transitive, so if some port is shorted to two other ports, all three ports are shorted. A switch is programmed by specifying its shorts. All `switch_` types are declared without arguments and initially contain no shorts. The `clear()` method has no arguments and clears all shorts from the `switch_` lvalue to which it is applied.

5.4.16.1 r (Restricted) and l (Local)

The `short()` method adds a short to the programming of the switch lvalue to which it is applied. It has one or two arguments and returns no result. If the `switch_` to which `short()` is applied has the "r" modifier, then `short()` may only be invoked with one argument; otherwise, it may be invoked with either one or two. The arguments to `short()` are `direction_` types, but which `direction_` types are allowed depend on the architecture identifier of the `switch_` object to which `short()` is applied. The `direction_` type may include the "d" modifier if that architecture identifier supports the "d" modifier of "aC". Similarly, it may include the "l" modifier if the architecture identifier supports the "l" modifier of "T". Each `direction_` argument to `short()` specifies one of the switch's ports, either locally or globally. The port specified is the one through which communication would occur if that `direction_` value were used with adjacent communication. If `short()` is invoked with one argument, it shorts the specified port to the opposite port. (I.e., the port going in exactly the opposite direction.) If `short()` is invoked with two arguments, it shorts the two specified ports together.

When the "l" modifier of "T" is present in the architecture identifier of the switch to which `short()` is applied, `short()` only modifies the programming of switches in enabled PEs. In this case, `clear()` also operates only on switches in enabled PEs; however, declaration of a switch creates a program with no shorts in all PEs, enabled and disabled. When this modifier is not present, only a single switch program is being formed and the enable status of PEs is not relevant.

5.4.16.2 c (Connect Through Switch)

The `short_self()` method has one argument, which may have exactly the same types as the arguments to `short()`. `short_self()` shorts the switch's "PE" port (i.e., its connection to the PE) to the switch port specified by the argument. `short_self()` may be applied to a `switch_` object which has the "c" modifier of "T" in their architecture identifier.

The `short_self()` method may also be applied to a `switch_` object which does not have the "c" modifier of "T" in its architecture identifier. However, in this case the action is more limited; the `clear()` method is applied before `short_self()`. Any later application of `short()` undoes the action of `short_self()`. This use of `short_self()` specifies that the PE is to be connected to the communication port, and the PE's switch disconnected. Invoking `short()` specifies that the PE is to be disconnected from the communication network, and its switch connected; it also begins reprogramming the switch.

5.4.16.3 o ("Or" Combine)

When the "o" modifier of "T" is present, it can be specified that multiple values sent to the same set of PEs be combined with the bitwise or operation. This is specified by appending `_with_or` to the name of the `send()` method. It goes after `_wrap`, if `_wrap` is also used.

5.4.17 Local Addressing (L)

5.4.17.1 lL (Limited)

Limited local addressing involves the use of a special set of `simd_` array types. These types are named like ordinary `simd_` types, but have the string `_arr` appended to their name. These `simd_` array types have one additional declaration argument, which is the number of array elements to create. The array consists of that number of identical `simd_` objects, just like the one that would have been created if the `_arr` and third argument had been omitted. The virtual machine on which the `simd_` array exists must have an architecture identifier that supports the "lL" feature.

These `simd_` arrays may only be used in the ways described here. The `bits()` method may be applied to a `simd_` array, and returns the same value as if it had been called on an element of the array. The `elements()` method may be applied to a `simd_` array; it takes no arguments and returns the number of elements in the array (an unsigned scalar value). The `elem()` method is the primary way to use a `simd_` array. It has one argument, which is normally an unsigned integer `simd_` value. (The argument may also be an unsigned scalar value, in which case it is converted to an unsigned integer `simd_` value and used as if it had been `simd_`.) The argument must be on the same virtual machine as the `simd_` array to which it is applied. This argument identifies in each PE, enabled or not, the array element to be accessed in that PE. The return type of `elem()` depends on the way it is used. If it is used where a `simd_` rvalue may be used, it returns a `simd_` rvalue of the same type as the array's elements and containing the selected element's value in each PE (enabled or not). If it is used as the left argument to the plain assignment operator (=), then in each enabled PE the rvalue being assigned is copied into the selected element of the array. If the argument to `elem()` contains in any enabled PE a number greater than the number of elements in the array, the result is undefined.

5.4.17.2 uL (Unlimited)

Unlimited local addressing involves the use of a special set of `simd_` pointer types. These types are named like ordinary `simd_` types, but have the string `_ptr` appended to their name. These `simd_` pointer types have only one declaration argument; they omit the second argument of ordinary `simd_` declarations. The virtual machine on which the `simd_` pointer exists must have an architecture identifier that supports the "uL" feature.

These `simd_` pointers may only be used in the ways described here. The `set()` and `ref()` methods are the primary ways to use a `simd_` pointer. The `set()` method has one argument, which may have any of several types. If the argument is a `simd_` lvalue, then in each enabled PE the pointer is set to point to that `simd_` lvalue. If the argument is a `simd_` array to which the `elem()` method is applied, then in each enabled PE the pointer is set to point to the array element selected by the argument to `elem()`. If the argument is a `simd_` pointer, then in each enabled PE the pointer to which `set()` is applied is set to point to whatever the argument pointer points to. If the argument is the null pointer (`NULL`), then in each enabled PE the pointer is set to the null pointer. In any case, the argument must be on the same virtual machine as the pointer, and the basic type of the argument must be the same as the basic type of the `simd_` pointer. No other argument types are allowed. `set()` returns no value.

The `ref()` method has no arguments. Like the `simd_` array `elem()` method, the way it is used determines its action and return type. If `ref()` is applied to a `simd_` pointer where a `simd_` rvalue may be used, it returns a `simd_` rvalue of the same basic type as the `simd_` pointer to which it is applied. If a the pointer has a null value in some enabled PE, the result is undefined. If it is used as the left argument to the plain assignment operator (=), then in each enabled PE the rvalue being assigned is copied into the location pointed to by the `simd_` pointer in that PE. If a the pointer has a null value in some enabled PE, the result is undefined.

5.4.17.3 c (Communication with Local Addressing)

5.4.17.3.1 Intra-Machine Communication Communication with local addressing, provided by the "c" modifier of the "L" feature, is specified by replacing the first argument of any of the various legal forms (except cut-through communication) of `send()` and `fetch()` methods. When limited local addressing is provided, this first argument may be replaced by a pair of arguments: (1) a `simd_` array value (which must be an lvalue for `send()`), and (2) an unsigned integer `simd_` value to be used as an index into the array. The index value in each PE initiating communication is sent to the PE with which it is communicating; there, the index is applied to the array to address the data to use. The index value must be on the same virtual machine as the `simd_` value to which `send()` or `fetch()` is applied. The `simd_` array must also be on this virtual machine, unless one of the following conditions is met: (1) global communication is being used; (2) the two virtual machines have the same labeling in their architecture identifiers and were declared with the same argument values.

When unlimited addressing is also supported, the normal first argument of the various `send()` and `fetch()` methods may be replaced by a `simd_` pointer. The pointer value in each PE initiating communication is sent to the PE with which it is communicating; there, the pointer value is used to address the data to use. The pointer value must be on the same virtual machine as the `simd_` value to which `send()` or `fetch()` is applied.

When the `send()` or `fetch()` method specifies collision resolution, collisions are only considered to occur when multiple PEs communicate with the same PE using the same addressing value.

5.4.17.3.2 Inter-Machine Communication ("gC" only) Non-trivial communication between different virtual machines is only supported when the architecture identifier of the machine on which communication is initiated supports global communication ("gC"). This section describes how to specify local addressing when it is also supported by the architecture identifier of that initiating virtual machine. For limited local addressing, the method has already been alluded to. Simply use a `simd_` array argument to `send()` or `fetch()` which is on a different virtual machine than the `simd_` value to which the method is applied. The index value must be on the same virtual machine as the value to which the method is applied, but must contain valid index values for the array (which is on another virtual machine).

For unlimited local addressing, the invocation of the `send()` and `fetch()` is unchanged. However, the pointer is allowed to be set differently. The argument to the `set()` method may be on a different virtual machine than the pointer itself. A pointer set in this way may only be used for communication with local addressing; it may not be used for ordinary local addressing within a single virtual machine.

5.4.18 Reduce (R)

A set of methods with names of the form `reduce_OP()` may be applied to any `simd_` lvalue to specify reduction operations. In the case of global reduction only, described in a following paragraph, `reduce_OP()` may also be applied to a `simd_` rvalue. The legal values of `OP` depend on the combining operation modifier to the reduce feature of the `simd_` values's architecture identifier. The legal values for each architecture modifier are as follows.

f `first`.

o `or`, `and`.

x `xor`.

m `max`, `min`, or an "f" operator.

s `add`, or an "m", "x", or "o" operator.

p `mul`, or an "s" operator.

The meanings of these operators are described in detail in the taxonomy. The type of reduction operation is specified by the presence and type of argument to the `reduce_OP()` method.

5.4.18.1 d*R (Dimensioned)

If there is an argument to the `reduce_OP()` method, it must have one of two types. The argument may be an unsigned scalar value, which specifies a dimension along which to reduce. This single-dimensioned reduction is legal when the `simd_` lvalue's architecture identifier has the "d" or "m" prefix to its "R" feature. Single-dimensioned reduction groups PEs by coordinate in the other dimensions, so the PE labels of each group differ only in that dimension. The elements of the `simd_` value in the enabled PEs of each group are combined with `OP`. Then every PE in each group, enabled or not, receives a copy of its group's combined value. This combined value is the `reduce_OP()` method's return value. It is a `simd_` rvalue of the same basic type as the `simd_` lvalue to which the method is applied.

5.4.18.2 m*R (Multi-dimensioned)

The other legal type for `reduce_OP()`'s single argument is a pointer to an array of scalar unsigned values. The array must contain one value for each dimension of the `simd_` lvalue to which the method is applied. This use of `reduce_OP()` performs multi-dimensioned reduction. Multi-dimensioned reduction groups PEs by coordinate in the dimensions for which the corresponding argument array value is zero. The labels of PEs in each group therefore differ only in the dimensions for which the corresponding argument array value is non-zero. The elements of the `simd_` value in the enabled PEs of each group are combined with `OP`. Then every PE in each group, enabled or not, receives a copy of its group's combined value. This combined value is the `reduce_OP()` method's return value. It is a `simd_` rvalue of the same basic type as the `simd_` lvalue to which the method is applied.

5.4.18.3 g*R (Global)

If there is no argument, then global reduction is performed. This is legal when the `simd_` lvalue's architecture identifier has the "g" prefix to its "R" feature. Global reduction returns a scalar value of the same basic type as the `simd_` value to which it is applied. This returned value is obtained by combining the elements of the simd_ value in all enabled PEs, using the OP operator.

It is particularly useful and common to use the result of reduce_or() as a logical value. The method any() provides exactly that; it performs a global reduce_or(), and returns one if the result was non-zero and zero if the result was zero. It is legal wherever reduce_or() is, and returns the same type result. A special related method called any_enabled() is also provided. It is legal whenever any() is. It has no arguments. It returns a scalar unsigned value which is either zero or one. any_enabled() returns one if virtual machine holding the simd_ value to which it is applied has at least one PE enabled; it returns zero otherwise.

5.4.18.4 Flow Control with "goR"

Section 5.4.8 presented an IF ELSE ENDIF construct which always executes its body's statements, even if no PEs are enabled. The any_enabled() method may be used to write a similar structure which only executes its body's statements when at least one PE is enabled.

```
IF ( <simd expr> )
  if (<simd expr>.any_enabled()) {
    <statements>
  }
ELSE
  if (<simd expr>.any_enabled()) {
    <statements>
  }
ENDIF
```

Porta-SIMD gives the programmer the flexibility to use either construct when both are supported by the target architecture. And it provides flow control even when the "goR" feature is not supported.

The "goR" feature makes possible a second kind of flow control. A WHILE construct is written as follows.

```
WHILE ( <simd expr> )
    <statements>
ENDWHILE
```

<simd expr> represents an expression which evaluates to a value of a simd_ type. <statements> may be any number of Porta-SIMD statements.

The <simd expr> is evaluated before execution of the statement, and any enabled PEs which compute a zero value for it are disabled. After this, if <simd_expr>.any_enabled() is true then the statement is executed. This process is repeated until any_enabled() returns false. Once a PE is disabled, it remains disabled until the entire WHILE construct has performed all iterations and completed its execution. After any_enabled() returns false, all PEs disabled by the WHILE construct are enabled. The enable status of all PEs is therefore exactly as it was immediately before the WHILE construct began execution. Execution then resumes immediately after the WHILE construct.

5.4.19 Scan (S)

A set of methods with names of the form scan_OP() may be applied to any simd_ lvalue to specify reduction operations. The legal values of OP depend on the combining operation modifier of the scan feature of the simd_ values's architecture identifier. The legal values for each architecture modifier are the same as those described for reduce in the preceding section. The type of scan operation is specified by the presence or absence of an argument to the `reduce_OP()` method.

5.4.19.1 d*S (Dimensioned)

If there is an argument to the `reduce_OP()` method, it must be an unsigned scalar value, which specifies a dimension along which to scan. This single-dimensioned scan is legal when the `simd_` lvalue's architecture identifier has the "d" prefix to its "S" feature. Single-dimensioned scan groups PEs by coordinate in the other dimensions, so the PE labels of each group differ only in that dimension. The elements of the `simd_` value in the enabled PEs of each group are combined with `OP` in a prefix operation that computes the a result in each enabled PE. This partial result is the combination of its own element with all elements in enabled PEs in its group with a smaller coordinate in the dimension in which their labels differ. This partial result in enabled PEs is the `scan_OP()` method's return value. It is a `simd_` rvalue of the same basic type as the `simd_` lvalue to which the method is applied. It's value in disabled PEs is undefined.

5.4.19.2 g*S (Global)

If there is no argument to the `reduce_OP()` method, then a global scan operation is performed. Global scan is legal when the `simd_` lvalue's architecture identifier has the "g" prefix to its "S" feature. The elements of the `simd_` value in all enabled PEs are combined with `OP` in a prefix operation that computes the a result in each enabled PE. This partial result is the combination of its own element with all elements in enabled PEs with a smaller PE number. This partial result in enabled PEs is the `scan_OP()` method's return value. It is a `simd_` rvalue of the same basic type as the `simd_` lvalue to which the method is applied. It's value in disabled PEs is undefined.

5.4.20 Input (I)

5.4.20.1 I (Input)

The method `read()` may be applied to any `simd_` lvalue whose architecture identifier contains "I". `read()` has a single argument, an implementation dependent file specifier. `read()` copies the next value from the file into the `simd_` lvalue, copying the data into all PEs regarless of enable context. The format of the data is implementation dependent, but must be that used by `write()`. `read()` returns an integer scalar value which is non-negative if and only if the operation succeeded.

5.4.21 Output (O)

5.4.21.1 dO (Display Output)

The method `display()` may be applied to any `simd_` value whose architecture identifier contains or subsumes "dO". `display()` has a single optional argument, an implementation dependent destination specifier. If no argument is given, an implementation dependent default value is used. `display()` copies the `simd_` value to the destination, copying the data in all PEs regardless of enable context. The format of the data is implementation dependent, and need not be the same as that used by `read()` and `write()`. `display()` returns an integer scalar value which is non-negative if and only if the operation succeeded.

It is intended that the destination specifier encode information such as the display device and the component of the image that the `simd_` value represents. Typical components might be: "red", "green", "blue", "mapped-color", "rgb", "grayscale", "black-and-white".

5.4.21.2 O (Output)

The method `write()` may be applied to any `simd_` value whose architecture identifier contains "O". `write()` has a single argument, an implementation dependent file specifier. `write()` appends the `simd_` value to the file, copying the data in all PEs regarless of enable context. The format of the data is implementation dependent, but must be understood by `read()`. `write()` returns an integer scalar value which is non-negative if and only if the operation succeeded.

5.4.22 Get (G)

5.4.22.1 rG (Restricted Get)

The method `get()` may be applied with no argument to any `simd_` value whose architecture identifier contains or subsumes "rG". The result is a scalar value of the same basic type as the `simd_` value to which `get()` is applied. The value is the value of that `simd_` value in some arbitrary PE.

5.4.22.2 G (Get)

The method `get()` may be applied with one argument to any `simd_` value whose architecture identifier contains or subsumes "G". The result is a scalar value of the same basic type as the `simd_` value to which `get()` is applied. The value is the value of that `simd_` value in the PE specified by the argument. The argument is an unsigned scalar value which is used as a PE number. The argument value must be less than the number of PEs in the `simd_` value's virtual machine.

5.5 Potential Language Enhancements

I believe Porta-SIMD's portability, power, expressiveness and type safety are excellent. However, I do see some ways to improve the language. They all concern more concise, natural, and aesthetic ways of expressing common operations.

Communication between the host and PEs, now expressed with `set()` and `get()` methods, could be written with square braces (`[]`). Limited local addressing could also be written much more naturally by using square brackets to index into `simd_` arrays, instead of using the `elem()` method.

Unlimited local addressing would be improved by making the use of `simd_` pointers look more like the use of scalar pointers. One or more of the pointer-related operators (`&`, `*`, `->`, `.`) could be used for this purpose.

Communication could be expressed much more compactly if square brackets or parentheses were used in place of some forms of the `send()` and `fetch()` methods. There are almost certainly too many variations of these methods to use operator notation for all of them, and especially to remember what all those operators mean. But using operators for some of the more common cases would probably improve program readability. Square brackets seem more natural to me, but it may be ambiguous or confusing to use them for communication in addition to the other uses already planned. Parentheses are a reasonable alternative. Pointer syntax using `->` and directions might even be useful.

Each of these potential enhancements is likely to improve the appearance and readability of Porta-SIMD programs. Determining the best design for these new features will require substantial thought and experimentation, particularly rewriting programs using possible language designs to see how the features work out in practice. While such work is important to the evolution of Porta-SIMD, these language enhancements are all cosmetic. They are not expected to yield new insights concerning optimal portability.

Chapter 6

Implementing an Optimally Portable Language

Implementing an optimally portable language generally requires more work than implementing a less portable language. However, most of the additional work involves either analysis and checking to enforce architectural limits, or provision of restricted forms of the language's more powerful features. There is no reason I can find why a high-quality implementation of an optimally portable language should not provide at least as efficient execution of application programs as a high-quality implementation of any other high-level SIMD language. However, the technology used in the prototype Porta-SIMD implementation places severe limits on that implementation's completeness and efficiency. While C++ `#include` files and libraries were an excellent tool for producing early prototypes to help Porta-SIMD's design evolve rapidly, it is not an appropriate tool for developing a complete or highly efficient implementation. As a result, only an important but relatively small subset of Porta-SIMD has been implemented.

Chapter 5 begins by discussing in detail the requirements for designing an optimally portable language, as defined in chapter 2. Similarly, the first section of this chapter discusses the requirements for implementing an optimally portable language, and some of the alternatives available for doing such an implementation. That is followed by a description of the prototype implementation of Porta-SIMD, the knowledge gained from building this prototype, and the experience of Porta-SIMD users. Some example programs are included. The final section discusses the theoretical and practical performance that can be expected from a high-quality implementation of an optimally portable SIMD language, as well as the performance actually achieved by the prototype implementation of Porta-SIMD.

6.1 Requirements for an Optimally Portable Language

There are three requirements for optimally portable languages. The first, making each program specify its target architecture, is primarily a language design issue. It is an implementation issue only to the extent that the implementation must be able to parse the language. However, the other two requirements for an optimally portable language have a substantial impact on its implementation. They are discussed in the next two subsections.

6.1.1 Provide Architectural Features

When all the variations due to various interacting architectural features are counted, Porta-SIMD defines a large number of language features. Other optimally portable SIMD languages will share this characteristic, because it reflects the fact that several architectural features in chapter 3's taxonomy modify each other. This leads to a combinatorial explosion of distinct operations. For example, there are well over 100 variations of the `send()`

method. The number of distinct architectures, and hence Porta-SIMD architecture identifiers to support, is even larger, since even features which do not interact with each other can be independently selected for use in an architecture identifier. There are over 2 billion legal architecture identifiers. Providing large numbers of language features (operations) and architecture identifiers is the requirement which most distinguishes the implementation of an optimally portable SIMD language from that of an ordinary SIMD language.

6.1.1.1 Many Operations

There are several ways an implementation of an optimally portable language may deal with the large number of operations. In a few cases, it may be possible to implement a smaller but equivalent set of features. For example, the architecture identifier `2aC` is equivalent to `2adwlCmWrF`, but specifies many fewer communication operations. However, in many cases there is no equivalent architecture with significantly fewer operations. This is the case for both the CM-2 and MP-1 architectures.

The only way to implement all the required operations using C++ `#include` files and libraries is to list and define them all exhaustively. Each distinct operation must be defined separately for every `simd_` type. This is an important limit of the implementation method: it defines and implements operations, not characteristics of operations.

The natural way for a compiler to handle the many variations on operations is to parse the operation name and arguments to find the selection made for each dimension of variation. Handling the variations this way is much more tractable than enumerating them. Having recognized the components of the operation's specification, the compiler might generate code in one of two ways. It might build code for exactly the operation specified, by combining code which implements the various features combined in the operation. Or it might have built-in implementations for the most general and most common operations, and use a special case of a more general operation if an exact match is not available for the specified operation.

6.1.1.2 Many Architectures

An implementation of an optimally portable language for a SIMD computer must provide architecture identifiers representing every architecture subsumed by that computer's architecture. This is only a handful for a basic architecture like Pixel-Planes, but can be a very large number for architectures with many optional features. The CM-2 subsumes over one million architecture identifiers.

It is impractical for the current prototype implementation of Porta-SIMD to provide large numbers of architecture identifiers. Because each architecture identifier may appear in the names of several types, an implementation based on C++ `#include` files and libraries must contain many definitions for each architecture identifier it supports. The real problem is that the implementation technology can only deal with architecture identifiers as distinct entities, rather than as sets of architectural features. This is another aspect of the limit of this implementation method noted in the previous section: it defines and implements architectures, not architectural features.

Fortunately, compilers are not limited in this way. They can treat architecture identifiers in the natural way, as a set of features formed by selecting at most one from each of twelve categories. So the number of possible architecture identifiers should not be a significant problem for compiler-based implementations of optimally portable languages.

6.1.2 Enforce Architectural Limits

Enforcing the architectural limits each program specifies for itself presents some of the same issues discussed in the previous section. Enforcing a ban on language features not supported by the program's target architecture is closely related to providing those features that are supported by that architecture.

Again, the C++ library method used in the prototype Porta-SIMD implementation has severe limits. It prohibits the use of language features by not defining them. Since it is unable to define all the required architecture identifiers, it incorrectly bans those it is not able to define. As before, a compiler need not have this difficulty, and should be able to enforce architectural limits correctly.

6.2 Porta-SIMD Implementation Results

A prototype implementation of Porta-SIMD has been performed. It implements a subset of the language, and runs on Pixel-Planes, the Connection Machine, and a sequential computer simulating a SIMD architecture.

This section reports on the prototype implementation Porta-SIMD. The implementation evolved with my ideas and the language design. Its present structure is briefly reviewed, and some of the difficulties inherent in the implementation approach are described. The current status of the implementation is reported for each computer supported, and the implemented subset of the language is defined. Three existing SIMD programs were rewritten in Porta-SIMD and are presented in this section. Their performance is reported in section 6.3.4. The details of compiling and running a Porta-SIMD program are described, and the comments of Porta-SIMD users are reported. Finally, the effort required to port the prototype implementation is described.

6.2.1 History

Porta-SIMD's prehistory began in late 1987 when I began experimenting with C++ as a tool for data parallel programming at Dr. Brooks's suggestion. By the end of the year I had the beginning of the concept of optimal portability, and a tiny beginning on Porta-SIMD. In March 1988, Porta-SIMD provided the single parallel type `simd_int` and the C assignment operators (e.g., `+=`, `|=`, etc.), and ran on PxPl4 (Pixel-Planes 4) and sequential machines.

Next, I added "virtual memory" to the sequential version, in preparation for the off-chip backing store memory of PxPl5 (Pixel-Planes 5). This feature allows programs to use the entire off-chip memory space, with the implementation transparently moving data into on-chip memory to operate on it as needed. I had the opportunity to visit Argonne National Laboratory's ACRF (Advanced Computer Research Facility), where I ported the then-current implementation of Porta-SIMD to the CM-2 (Connection Machine, model 2) in less than a week. I presented a paper on my work at Frontiers '88 [Tuck88], and implemented the remaining C operators for `simd_int` by the end of the year.

In early 1989, I implemented two additional parallel types: `simd_unsigned` and `simd_bool`. This meant handling type conversions for the first time, which required a lot of work. I also wrote a User's Manual for Porta-SIMD [Tuck89]. By May, I had also ported Porta-SIMD to the PxPl5 simulator. The next task was to add support for multiple virtual machines of different sizes. That was largely completed in July, along with a complete rewrite of the memory management system to support it. The improved memory management was also needed to support the different memory model used by Paris with release 5 of the CM-2's system software.

At this point, C++ 2.0 was released and I revised the implementation's class structure to take advantage of its new features, including multiple inheritance. This was important preparation for supporting multiple architecture identifiers effectively. Compiler problems slowed porting of the new features to the CM-2, but the CM-2 version was up to date by the end of 1989. The CM-2 at CMNS (the Connection Machine Network Server) was used for this port and subsequent work on the CM-2.

Work in 1990 has focused on implementing optional architectural features, the most important of which are adjacent and global communication and local addressing.

6.2.2 Structure

The prototype implementation of Porta-SIMD consists of a few preprocessor macro definitions for flow control and several families of C++ classes which implement the types described in the language definition. There are two main sets of classes: the programmer interface classes which implement the parallel data types, and the hardware interface classes which implement virtual machines. Although Porta-SIMD programs may declare and use virtual machines, they directly use only a tiny part of the capabilities of virtual machines. The real use of virtual machines is by the parallel data type classes, which use virtual machines to do all parallel computation.

These two sets of classes are similarly structured. Each has a base class which implements the null architecture identifier. Classes are derived from this class to implement individual architectural features. Additional layers of classes are then derived with multiple inheritance, each deriving from the feature classes of the architecture identifier it implements. Since some features interact, there can be significant amounts of code in classes above the bottom "feature" layer, and higher layers may inherit from these combined-feature nodes. The virtual machine family of classes has names of the form `simd_ARCHID_mach`, where `ARCHID` is an architecture identifier. The parallel data family of classes is actually three very similar families: `simd_ARCHID_unsigned`, `simd_ARCHID_int`, and `simd_ARCHID_bool`.

The parallel data families all derive from from the `simd_ARCHID_var` family, which encapsulates the shared characteristics of all data types, including access to memory management classes. Both the `_var` classes and the memory management classes are invisible to Porta-SIMD programs.

The reader interested in more details of the class structure and the public and private interfaces of these classes is invited to study appendix A. It contains the C++ header files which declare these classes.

6.2.3 Implementation Difficulties

Although using C++ `#include` files and libraries to implement Porta-SIMD allowed early rapid prototyping and avoided writing a compiler, it also caused some problems. Section 6.1 has already pointed out that it is impractical to implement the full Porta-SIMD language with this technology because of the large number of language features and architecture identifiers. The two other major difficulties have been with conversions among `simd_` types, and with compiler bugs.

C++ supports user-defined implicit type conversions. Unfortunately, these have the wrong semantics for conversions between `simd_` types. The problem stems from the fact that the length of the elements of a `simd_` type is not part of the type name, but an argument to the declaration. In designing Porta-SIMD, I felt it was important to be able to specify integers with any number of bits in order to make the most effective use of limited PE memory. But making each size integer a different type would have multiplied the number of types and seriously complicated implementation. Making the length invisible to the type system means each operator must be able to handle arguments with different lengths, which is acceptable. But because the length is not visible to the C++ type system, implicit conversions do not change lengths.

An example will show the conversion problem.

```
simd_int signed16(NULL, 16), signed32(NULL, 32);
simd_unsigned uns16(NULL, 16), uns32(NULL, 32);
uns32 = signed16;
signed32 = uns16;
```

C++ invokes the conversion operation, which does not change the length and so does nothing. Then the assignment operator is given two operands of the same type but different lengths. The result is that the value of `signed16` is zero-extended, and the value of `uns16` is sign-extended. This is the opposite of the correct behavior, which is to sign-extend the signed value and zero-extend the unsigned value.

In order to implement type conversion correctly, it was necessary to define multiple versions of every operator, one version for every possible combination of argument types. Multiplying the number of operations this way created an unwieldy number of small but distinct subroutines in both the machine-independent and machine-dependent layers of the implementation.

Another difficulty with user-defined implicit conversions in C++ caused the limitations in cast operations described in section 5.4.6.2. C++ provides no mechanism for specifying which of several possible user-defined conversions to use. Instead, it requires that at most one user-defined conversion be applicable in each situation.

Although C++ is an excellent language, it is still evolving and its compilers are not yet as mature as those for older languages like C. As the prototype Porta-SIMD implementation evolved, it repeatedly uncovered compiler bugs. In June 1989 my code reached a point where the AT&T C++ 1.2 compiler got internal errors when attempting to compiling my code. I was not able to find a work-around, but was able to continue development using the GNU G++ compiler. I soon found a bug in it, too, but the bug only affected Sun-4s, so I was able to continue work using Sun-3s; the bug was fixed in October.

UNC got C++ 2.0 from AT&T in December 1989, so I had two working C++ compilers again. I immediately found a pair of complimentary bugs, one in each compiler, but a work-around, too. But by the end of January I had found another bug in the AT&T 2.0 compiler; this one caused it to generate illegal C code as output. I was not able to find a satisfactory work-around, so I thereafter used only the GNU compiler. In February I found a bug in the GNU compiler which halted work. A fixed version was available by the beginning of March.

Each compiler bug was a drain of time and energy while I identified the problem, reported it, and tried to work around it.

6.2.4 Status

The prototype implementation of Porta-SIMD implements a subset of Porta-SIMD. This subset includes the integer and boolean parallel types, but no parallel floating-point types. The following architectural features are implemented: 2-D adjacent communication (`2aC`), global communication (`gC`) with write collisions handled by selection (`sW`) and fetch collisions handled by replication (`rF`), limited local addressing both alone and during communication (`lcL`), global reduction with "and" and "or" (`goR`), display output (`dO`), and unlimited PE to host "get" I/O (`G`). All the base architectural features common to all SIMD machines and the optional architectural features `2aCG` are tested, working, and in use. However, the features `gCsWrFlcLgoRdO` are still in the debug and test phase.

Naturally, the CM-2 version supports all these features; the PxPl4 version supports no optional features except `2dO`; and the PxPl5 simulator version supports features `2dOG`. The versions for PxPl4 and the PxPl5 simulator do not support multiple virtual machines. The sequential version currently simulates the same features as the PxPl5 simulator version, though the simulators are completely unrelated.

The PxPl4 version implements the Porta-SIMD language in an earlier stage of its evolution. This is because there is no port of C++ 2.0 to the PxPl4 Graphics Processor on which PxPl4 programs run. Code for the PxPl4 version of Porta-SIMD was therefore frozen before the the Porta-SIMD implementation was moved from C++ version 1.2 to 2.0.

The prototype implementation of Porta-SIMD consists of declarations in `.h` (`#include`) files, definitions in `.C` (C++ source) files, compilation rules in `make` files, and test programs used solely to exercise the implementation. There are 103 `.h` files containing a total of over 8500 lines. Of these, 72 contain declarations used directly by user programs; the rest are directly used only by implementation source files. Appendix A consists of all the `.h` files. The `.h` files are `#include`'ed by 98 `.C` files, which contain a total of over 17300 lines and are stored in 27 directories. The prototype implementation is compiled and built into libraries using 33 `make` files with a total of over 3400 lines. A test suite comprising 72 `.C` files and over 8300 lines of code was used to test the prototype implementation. Together, these four groups of files contain over 37000 lines. These line counts are for all lines in the respective files, including comments and blank lines. None of these statistics include source which was written and discarded as part of the implementation's evolution.

6.2.5 Example Programs

This section presents some example Porta-SIMD programs. Each was already in use in the UNC Department of Computer Science when it was translated to Porta-SIMD. Some measurements have been made of the performance of these programs. Those results are reported in section 6.3.4.

The first program computes and displays the Julia set, a fractal pattern closely related to the Mandelbrot set. In fact, the main computation of these two sets is done identically; the only difference is in the initialization. Greg Turk originally wrote this program using the low-level C-callable "FB Macros" interface to PxPl4. He translated the core of this program to Porta-SIMD very early in Porta-SIMD's development, and provided valuable user feedback in the process. The program has since been updated to reflect Porta-SIMD's evolution. The Julia set program uses the `2` (2-D) and `dO` (display output) features, and runs on all versions of Porta-SIMD.

The Julia set is the set of points in the complex plane c at which, for a given complex constant z, $z = z^2 + c$ converges to a finite value. There is a distinct Julia set for every value of z. (The Mandelbrot set is the set of starting points in the complex plain z at which this equation converges when c is zero.) Approximations of these sets are normally computed by doing some finite number of iterations, during which most points not in the set reach a value which assures that they diverge to infinity.

The Julia set program is shown in figures 6.1–6.6. The first part, figure 6.1, contains declarations including program constants and global variables. Part 2, figure 6.2, shows the driver routine for the Julia set calculation. It initializes the points to be computed, calls the routine `iterate()` to perform each iteration, and colors and displays the result. Points not in the set are colored based on the number of iterations required to detect their assured divergence. A simple two-color scheme is used, based on whether the number of iterations was even or odd.

Part 3, figure 6.3, shows the `iterate()` routine, performs one iteration of the overall Julia set calculation. The precision with which the calculations can be carried out is limited

```
/* Julia in Porta-SIMD.  Greg Turk. */

#include <stdio.h>
#include <std.h>                /* atoi() */
#include "simd_2d0.h"

/* Image will be IM_SIZE x IM_SIZE */
#define IM_SIZE 35

/* specification of fixed-point size */
const int length = 10;          // total length
const int int_part = 3;         // bits to left of decimal
const int fract_part = length - int_part; // bits to right

/* other constants */
const int color_length = 8;

/* forward procedure declarations */
void compute_positions(float, float, float, float, float, float,
         float, float&, float&, float&, float&, float&, float&);
void check_overflow (simd_2_int& val, simd_2_bool& oflow);
void get_bit(simd_2_bool& dst, simd_2_int& src, int position);
void draw_julia(float, float, int);
void iterate(float, float);
float rnd(float, float);

/* globals */
simd_2d0_mach mach(IM_SIZE, IM_SIZE);

simd_2d0_unsigned color(&mach, color_length); // result image

simd_2_int  tbig(&mach, 2 * length);  // double sized temporary
simd_2_int  a(&mach,length);          // real and complex parts
simd_2_int  b(&mach,length);
simd_2_bool in(&mach);                // are we still in set?
simd_2_bool parity(&mach);            // parity of # iterations

float width = 4.0;

#ifdef pxpl4gp
short doneflag = 0; /* Flag to tell host when program ends. */
#endif

/* constants */
const float aspect = 1.0;
const float magic  = (1 << (length - 3));
```

Figure 6.1: julia.C — Julia Set Program (part 1).

```
/* Draw a Julia set. */
void draw_julia(float dx, float dy, int iters)
{
  float xa,xb,xc;
  float ya,yb,yc;

  /* compute (a,b) positions */
  compute_positions(0.0, 0.0, (float)IM_SIZE, (float)IM_SIZE,
                    0.0, 0.0, width, xa, xb, xc, ya, yb, yc);

  a.bilinear(xa, xb, xc);
  b.bilinear(ya, yb, yc);

  /* reset parity of number of iterations, and "in" flag. */
  parity = 0;
  in = 1;

  /* perform iteration */
  for (int i = 0; i < iters; i++)
    iterate (dx, dy);
  }

  /* outside Julia set alternates red and blue stripes */
  IF (parity)
    color = 0;
  ELSE
    color = 110;
  ENDIF

  /* inside Julia set is white */
  IF (in)
    color = 255;
  ENDIF
  color.display();
}
```

Figure 6.2: julia.C — Julia Set Program (part 2).

on PxPl4 by the machine's small memory (72 bits per PE). The code and its comments reflect some effort to minimize memory use in order to achieve the maximum precision (greatest fixed-point integer length) possible in the calculation. This is only necessary because of the limitations imposed on the prototype implementation by the fact it is not a compiler.

Part 4, figure 6.4, contains the routine which checks for overflow in the fixed-point format used by the program. It is used by `iterate()` as part of the test for divergence. Part 5, figure 6.5, presents a small main program to draw some Julia sets, and a support routine used by the overflow checking routine. Part 6, figure 6.6, shows two sequential support routines. One is used (in figure 6.2) as part of the initialization of points prior to beginning the iterative calculation. The other produces random coordinates which the main program uses in choosing Julia sets to draw.

The second program was also written by Greg Turk. It computes and displays the game of life. It originally ran on the MP-1 (MasPar), and was written in MPL. It uses the features `2aC` (2-D adjacent communication) and `G` (get). The parallel portions are shown in figures 6.7 and 6.8.

Figure 6.7 shows the declaration of the virtual machine and grid of cells to be simulated. It features the routine which computes a generation of the game of life, which was discussed in detail in section 5.3. Figure 6.8 shows support routines used by the main program to initialize the grid of cells. The `extern "C" {}` statements in figure 6.7 may be ignored. They make the subroutines they name callable from C as well as C++.

Figure 6.9 shows two main programs which use the life routines in the preceding two figures. The first simply runs the life simulation and prints the results of each iteration. The second invokes a graphical user interface to the game of life. Greg Turk wrote this interface in C using the X window system. The `extern "C" {}` statements make it possible to correctly make calls between the C user interface and the Porta-SIMD (C++) code shown.

The final example presented here is the parallel part of a program by Tim Cullip. It accepts a surface spaceified as an input image, where each intensity value in the image is the surface's elevation at that point. The program calculates the Intensity Axis of Symmetry [GauchPizer88] of the surface to detect features of the surface. It was written in MPL for the MP-1, and will run on the CM-2 version of Porta-SIMD when testing and debugging of the necessary features is complete. It uses some of the most powerful features of the MP-1 and CM-2 architectures. These include: `2aC` (2-D adjacent communication), `gCsWrF` (global communication with collision resolution: selection on write and replication on fetch), `lcL` (limited local addressing, including during communication), and `G` (get value from PE to host). This program is contained in figures 6.10– 6.17, and its internal structure is describe by its comments.

6.2.6 Using Porta-SIMD

The mechanics of compiling and linking a Porta-SIMD program are very similar to those for an ordinary C++ program. Each step requires only a minor addition.

When compiling and linking a Porta-SIMD program by hand, as opposed to using `make`, begin by defining the shell variable `PORTA_SIMD`. (At UNC, it should have the value `/home/common/tuck/simd`.) Using the `csh`, this is done with the following command.

```
setenv PORTA_SIMD /home/common/tuck/simd
```

The command which compiles each program source file into an object file must predefine two symbols using the `-D` compiler option. The first symbol identifies the type of SIMD hardware on which the program will run. This may be `seq` (to simulate parallel hardware using a sequential computer), `pxpl4` (to use PxPl4), `pxpl5sim` (to use the PxPl5 simulator), or `cm` (to use the CM-2). The second symbol identifies the host computer type. This

```
/* Perform one iteration. */
void iterate(float dx, float dy)
{
  simd_2_bool oflow(&mach);

  /* (a-b) * (a+b) (Use explicit tmps to conserve memory.) */
  tbig = a;
  tbig -= b;
  b += a;
  tbig *= b;

  /* check for overflow */
  check_overflow(tbig, oflow);
  IF (oflow) in = 0; ENDIF

  /* shift result right to get rid of extra fraction bits. */
  tbig >>= fract_part;

  /* Recompute original b; update a and put old a in tbig. */
  b -= a;
  { /* Create block so temp is freed as soon as possible. */
    simd_2_int temp = a;
    a = tbig;
    tbig = temp;
  }
  tbig *= b;  /* a * b */

  /* check for overflow */
  check_overflow(tbig, oflow);
  IF (oflow) in = 0; ENDIF

  /* shift to multiply by two, and check for overflow again */
  tbig <<= 1;
  check_overflow(tbig, oflow);
  IF (oflow) in = 0; ENDIF

  /* shift result right to get rid of extra fraction bits. */
  tbig >>= fract_part;
  b = tbig;     /* move new product into b */

  /* add in the shift factor (use block to free tmp quickly) */
  { a += magic * dx; }
  { b += magic * dy; }

  /* update parity of iterations */
  IF (in)
    parity = !parity;
  ENDIF
}
```

Figure 6.3: julia.C — Julia Set Program (part 3).

```
/* Check to see if abs(value) >= 2.
   Entry: value - value to check magnitude of
   Exit:  oflow - 1 if overflow, 0 if not
*/
void check_overflow(simd_2_int& value, simd_2_bool& oflow)
{
  simd_2_bool neg(&mach), bit(&mach)n;

  /* assume no overflow */
  oflow = 0;

  /* see if value is negative */
  neg = (value < 0);

  IF (neg)  // if negative
    get_bit(bit, value, fract_part * 2 + 3);
    IF (!bit)
      oflow = 1;
    ENDIF
    get_bit(bit, value, fract_part * 2 + 4);
    IF (!bit)
      oflow = 1;
    ENDIF
  ELSE      // if positive
    get_bit(bit, value, fract_part * 2 + 3);
    IF (bit)
      oflow = 1;
    ENDIF
    get_bit(bit, value, fract_part * 2 + 4);
    IF (bit)
      oflow = 1;
    ENDIF
  ENDIF
}
```

Figure 6.4: julia.C — Julia Set Program (part 4).

```
/* Main program. */
main(int argc, char *argv[])
{
  int i;
  int num = 50;

  if (argc > 1)
    num = atoi(argv[1]);

  /* Draw set at 4 particular points, then at random points. */
  draw_julia(0.0, 0.0, 50);
  draw_julia(0.245, 0.0, 50);
  draw_julia(-0.7761, 0.0, 50);
  draw_julia(-0.121, 0.755098, 50);
  for (i = 0; i < num; i++)
    draw_julia(rnd(-1.0, 1.0), rnd(-1.0, 1.0), 50);

#ifdef pxpl4gp
  doneflag = 1;
#endif
}

/* Extract one bit of a simd_int into a simd_bool.
   Entry: source   - source simd_int to extract bit from
          position - position of bit to get (LSB is position 1)
   Exit:  dest     - place to return bit
*/
void get_bit(simd_bool& dest, simd_int& source, int position)
{
  int mask = (1 << (position - 1));
  dest = mask & source;
}
```

Figure 6.5: julia.C — Julia Set Program (part 5).

```
/* Compute initial values of (a,b). */
void compute_positions(float x1, float y1, float x2, float y2,
  float cx, float cy, float width, float& xa, float& xb,
  float& xc, float& ya, float& yb, float& yc)
{
  y2 = y2;  // so C++ won't complain about y2 being unused
  float dx;
  dx = x2 - x1;

  /* compute value for a */
  xa = magic * width / dx;
  xb = 0.0;
  xc = -magic * (width / 2 - cx) - x1 * magic * width / dx;

  /* compute value for b */
  ya = 0.0;
  yb = -magic * width / dx / aspect;
  yc = magic * (width / 2 / aspect + cy) +
       y1 * -magic * width / dx / aspect;
}


/* Return a random number between min and max. */
float rnd(float min, float max)
{
  float r = (rand() & 0xffff) / (float)0xffff;
  return(r * (max - min) + min);
}
```

Figure 6.6: julia.C — Julia Set Program (part 6).

```
/* Life for the Maspar.  Greg Turk, February 1990.
 * Translated from MPL to Porta-SIMD by Russ Tuck.
 */
#include <simd_2aCG.h>
const unsigned rows = 32, cols = 64;
simd_2aCG_mach mach(cols, rows);
simd_2aCG_bool cell(&mach);     /* the life grid of cells */

#ifndef SIMPLE_MAIN  /* Make some routines C-callable. */
extern "C" { void compute_iterations(int iters); }
extern "C" { void cell_flip(int x, int y); }
extern "C" { void init_cells(); }
#endif

/*** Compute several generations of life. ***/
void compute_iterations(int iters)
{
  int i;
  const direction_aC E(0,1), W(0,-1), N(1,1), S(1,-1);
  simd_2aC_int tsum(cell.m_2aC());
  simd_2aC_int sum(cell.m_2aC());
  simd_2aC_int tmp(cell.m_2aC()); /* used in communication */

  for (i = 0; i < iters; i++) {

    /* count the living neighbors */
    tsum  = cell;
    tsum += tmp.fetch(cell, N);
    tsum += tmp.fetch(cell, S);

    sum   = tsum;
    sum  += tmp.fetch(tsum, E);
    sum  += tmp.fetch(tsum, W);
    sum  -= cell;

    /* compute new generation */
    IF (cell)
      IF (sum < 2 || sum > 3)
        cell = 0;
      ENDIF
    ELSE
      IF (sum == 3)
        cell = 1;
      ENDIF
    ENDIF
  }
}
```

Figure 6.7: life.C — Game of Life Program (part 1).

```
/*** Flip the state of a cell. ***/
void cell_flip(int x, int y)
{
  simd_2_int tmp(cell.m_2());

  IF (tmp.pe_coord(0) == x && tmp.pe_coord(1) == y)
    IF (cell == 0)
      cell = 1;
    ELSE
      cell = 0;
    ENDIF
  ENDIF
}

/*** Create a single glider. ***/
void create_glider(int x, int y)
{
  cell_flip (x, y);
  cell_flip (x, y+1);
  cell_flip (x, y+2);
  cell_flip (x+1, y+2);
  cell_flip (x+2, y+1);
}

/*** Create the decathlon (ten cells in a row). ***/
void create_decathlon(int x, int y)
{
  int i;

  for (i = 0; i < 10; i++)
    cell_flip (x + i, y);
}

/*** Create a glider and decathlon. ***/
void init_cells()
{
  cell = 0;
  create_glider (10, 10);
  create_decathlon (30, 16);
}

/*** Turn on one cell. ***/
void cell_on(int x, int y)
{
  cell.set(label_2(cell.m_2(), x, y), 1);
}
```

Figure 6.8: life.C — Game of Life Program (part 2).

```
#ifdef SIMPLE_MAIN
/*** Simple main program to demo game of life. ***/
#include <stdio.h>
#include <std.h>     /* getenv(), strtol() */
void print_cells() {
  for (int row=0; row < rows; row++) {
    for (int col=0; col < cols; col++)
     printf("%c",cell.get(label_2(cell.m_2(),col,row))?'X':' ');
    printf("\n");
  } printf("\n");
}

int main() {
  /* Get # of iterations from environment var LIFE_ITERS. */
  unsigned life_iters = 10;  /* Default # of iters. */
  char *env_strp, *tmp_strp;
  char **check_strpp = &tmp_strp;
  if ((env_strp = getenv("LIFE_ITERS")) != NULL) {
    long tmp_val = strtol(env_strp, check_strpp, 0);
    if (*check_strpp == env_strp)
      printf("Ignoring non-int value of env var LIFE_ITERS.\n");
    else life_iters = tmp_val;
  }
  printf("Will compute %u iterations.\n", life_iters);

  init_cells();
  for (int i=0; i <= life_iters; i++){
    printf("Iteration %d:\n", i); print_cells();
    compute_iterations(1);
  }
  return(0);
}

#else
/*** Send cells back to host. ***/
extern "C" { void give_cells(int *host_cell); }
void give_cells(int *host_cell)
{
  int *cell_ptr = host_cell;
  for (int row=0; row < rows; row++) {
    for (int col=0; col < cols; col++) {
      *(cell_ptr++) = cell.get(label_2(cell.m_2(), col, row));
    }
  }
}
/*** Use X user interface (code not shown). ***/
extern "C" { int realmain(int argc, char *argv[]); }
main(int argc, char *argv[]) { return(realmain(argc,argv)); }
#endif
```

Figure 6.9: life.C — Game of Life Program (part 3).

```
/* surface.C by Tim Cullip.
 * Translated from MPL to Porta-SIMD by Russ Tuck.
 */
#include <stdio.h>
#include <simd_2gCsWrFlcLG.h>
#include <image.h>            /* UNC's "/usr/image" library. */

const GREYLEN 16;             /* Bits used for images */
const direction_aC E(dimen_x, 1), W(dimen_x, -1),
                   N(dimen_y, 1), S(dimen_y, -1);
const int Debug = 1;

/* surface(): treat input image as a surface, using the pixel
 * values as surface elevation, and calculate the Intensity
 * Axis of Symmetry of the surface to detect its features.
 *
 * At call: pix points to an xdim by ydim by 3 array of
 * GREYTYPE pixels.  These are arranged as three xdim by ydim
 * images in succession.  The first image is the input, and the
 * other two are undefined.  The other parameters control the
 * computation, and are input only.
 * At return: all three images pix points to are full.  They
 * contain respectively the original x and y coordinates and
 * original input value (intensity) of each output pixel.
 */
void surface(GREYTYPE *pix, int xdim, int ydim,
             GREYTYPE min, GREYTYPE max,
             double fw1, double fw2, double fw3,
             int MaxIteration, int SpreadFactor)
{
   /* Process weight arguments. */
   int w1 = 100*fw1;
   int w2 = 100*fw2;
   int w3 = 100*fw3;

   int i,j,x,y,i1,i2;
   plural GREYTYPE val;

   printf("Start dpu_setup(); dims: %d %d\n",xdim,ydim);
   printf("intensity range: %d %d\n",min,max);
   printf("weights: %d %d %d\n",w1,w2,w3);
   printf("iters,spread: %d %d\n",MaxIteration,SpreadFactor);
```

Figure 6.10: surface.C — Intensity Axis of Symmetry Program (part 1)

```
    /* Allocate machine and some variables on it. */
    simd_2gCsWrFlcLG_mach *machptr =
      new simd_2gCsWrFlcLG_mach(xdim,ydim);

    /* Input image will be loaded into I.
     * X, Y, and I are computed, and become the output data.
     */
    simd_2G_int X(machptr, GREYLEN);
    simd_2G_int Y(machptr, GREYLEN);
    simd_2G_int I(machptr, GREYLEN);
    /* b: binary image from intensity threshold */
    simd_2aCgoR_bool b(machptr);
    /* d: manhattan distance to a 0 in b (radius). */
    simd_2_int d(machptr, GREYLEN);
    /* r: array of values of d, one for each intensity. */
    int num_r = max-min+1;
    printf("num_r: %d\n",num_r);
    simd_21L_int_arr r(machptr, GREYLEN, num_r);
    /* R: array matching r, with true values wherever the r
     * value is a local maximum along either dimension. */
    simd_2gCsWrFlcL_bool_arr R(machptr, GREYLEN, num_r);
    simd_2_int F(machptr, GREYLEN);
    simd_2_int *D[9];
    for (i=0; i < 9; i++) {
      D[i] = new simd_2_int(machptr, GREYLEN);
    }

    /* Copy input image to I. */
    GREYTYPE *pix_ptr = pix;
    for (i=0; i < I.pes(); i++) {
      I[i] = *pix_ptr++;
    }
    printf("Input image is in parallel memory.\n");

    /* Do real work. */
    init_ias(X, Y);
    init_radius(I, b, d, r);
    converge_surface(X, Y, I, 0, D, F, R, min, max, r);
    MaxIteration = 10;
    converge_surface(X, Y, I, 1, D, F, R, min, max, r);

    /* Copy result data back to scalar array (*pix). */
    pix_ptr = pix;
    for (i=0; i < X.pes(); i++) { *pix_ptr++ = X[i]; }
    for (i=0; i < Y.pes(); i++) { *pix_ptr++ = Y[i]; }
    for (i=0; i < I.pes(); i++) { *pix_ptr++ = I[i]; }
    printf("Output data is in sequential memory.\n");
}
```

Figure 6.11: surface.C — Intensity Axis of Symmetry Program (part 2)

```
void init_ias(simd_2_int& X, simd_2_int& Y)
{
  X.pe_coord(dimen_x);
  Y.pe_coord(dimen_y);
}

void init_radius(simd_2_int& I, simd_2aCgoR_bool& b,
                 simd_2_int& d, simd_21L_int_arr& r)
{
  GREYTYPE Intensity,i,dist;
  GREYTYPE intensity_offset;
  simd_2aC_int tmp = d;

  for (Intensity = min; Intensity <= max; Intensity++) {
    /* first create the binary image inside the contours
     * and zero the distance function */
    IF (I >= Intensity)
      b = 1;
    ELSE
      b = 0;
    ENDIF;
    d = 0;
    /* now set up to iterate until all 1's are burned away */
    dist = 1;
    WHILE (b) {
      /* if I'm a 1 and any neighbor is a 0, record distance */
      IF (b)
        tmp = b;
        IF (!tmp.fetch(b, N)) d = dist; ENDIF;
        IF (!tmp.fetch(b, E)) d = dist; ENDIF;
        IF (!tmp.fetch(b, S)) d = dist; ENDIF;
        IF (!tmp.fetch(b, W)) d = dist; ENDIF;
      ENDIF;
      /* if my distance just got recorded, turn me off */
      IF (d == dist)
        b = 0;
      ENDIF;
      dist++;
    }
    intensity_offset = Intensity - min;
    r.elem(intensity_offset) = d;
  }
}
```

Figure 6.12: surface.C — Intensity Axis of Symmetry Program (part 3)

```
/* For each level of intensity, identify PEs at local maximum
 * radius along either dimension.  Each such PE is on a ridge.
 */
void RadiusRidge(GREYTYPE min, GREYTYPE max,
                 simd_2lL_int_arr& r, simd_2lL_bool_arr& R)
{
  simd_2aC_int p(r.m_2aC(), GREYLEN);
  simd_2aC_int tmp = p;

  for (int i_offset=0; i_offset <= max-min; i_offset++) {
    p = r.elem(i_offset);
    R.elem(i_offset) = 0;
    IF (((p > tmp.fetch(p, E)) && (p > tmp.fetch(p, W))) ||
        ((p > tmp.fetch(p, N)) && (p > tmp.fetch(p, S))))
      R.elem(i_offset) = 1;
    ENDIF;
  }
}

/* Calculate the energy at each point on the surface.
 * Energy decreases as radius increases (climbing a ridge),
 * but increases as continuity constraints are violated.
 */
simd_2_int SurfEnergy(simd_2_int& ix, simd_2_int& iy, int ind,
                      simd_2_int *D[], simd_2_int& F,
                      int SecondPass, simd_2aC_mach *m)
{
  simd_2_int x(m), y(m), z(m), rindex(m), pindex(m);
  simd_2aC_int xleft1(m), xleft2(m), xright1(m), xright2(m);
  simd_2aC_int xbottom1(m), xbottom2(m), xtop1(m), xtop2(m);
  simd_2aC_int yleft1(m), yleft2(m), yright1(m), yright2(m);
  simd_2aC_int ybottom1(m), ybottom2(m), ytop1(m), ytop2(m);
  simd_2_int su(m), sv(m);
  simd_2_int Xu(m), Xv(m), Xuu(m), Xvv(m), Xuv(m);
  simd_2_int Yu(m), Yv(m), Yuu(m), Yvv(m), Yuv(m);
  simd_2_int dist(m, 16);
  simd_2_int Energy(m, GREYLEN);
  simd_2_int retval(m, 32);

  x = ix;  y = iy;
  IF      (x < 0)     retval=10000;
  ELSE IF (x >= xdim) retval=10000;
  ELSE IF (y < 0)     retval=10000;
  ELSE IF (y >= ydim) retval=10000;
  ELSE {
```

Figure 6.13: surface.C — Intensity Axis of Symmetry Program (part 4)

```
z = I;
IF (F) {
  /* Find the radius value at the pixel location we want
     to move to, which was specified by caller. */
  dist.fetch(r, z, simd_label_2(x,y).pe_number());
  D[ind] = dist;
} ELSE { dist = D[ind]; } ENDIF

/* first pass: energy is radius at the specified x,y. */
Energy = 0.0;
if (SecondPass) {
  /* Look at neighbor values so can compute continuity
   * constraints. */
  ALL {
    /* Unconditionally copy x & y 1 PE in each direction. */
    xleft2.fetch(x, W);
    xright2.fetch(x, E);
    xbottom2.fetch(x, S);
    xtop2.fetch(x, N);
    yleft2.fetch(y, W);
    yright2.fetch(y, E);
    ybottom2.fetch(y, S);
    ytop2.fetch(y, N);
  } ENDALL;

  /* Conditionally save neighbor data while it's here. */
  xleft1   = xleft2;
  xright1  = xright2;
  xbottom1 = xbottom2;
  xtop1    = xtop2;
  yleft1   = yleft2;
  yright1  = yright2;
  ybottom1 = ybottom2;
  ytop1    = ytop2;

  /* Now conditionally fetch from neighbors copies of
   * their neighbor's X and Y, to get data 2 PEs away. */
  xleft2.fetch(xleft2, W);
  xright2.fetch(xright2, E);
  xbottom2.fetch(xbottom2, S);
  xtop2.fetch(xtop2, N);
  yleft2.fetch(yleft2, W);
  yright2.fetch(yright2, E);
  ybottom2.fetch(ybottom2, S);
  ytop2.fetch(ytop2, N);

  su = abs(xleft1 - xright1) + abs(yleft1 - yright1);
  sv = abs(xbottom1 - xtop1) + abs(ybottom1 - ytop1);
```

Figure 6.14: surface.C — Intensity Axis of Symmetry Program (part 5)

```
      /* Compute 1st and 2nd partial derivatives of x and y
       * with respect to u and v. */
      IF (su+sv < SpreadFactor) {
        Xu = (xright1 - xleft1)/2;
        Xv = (xtop1 - xbottom1)/2;
        Xuu = xright1 - x - x + xleft1;
        Xvv = xtop1 - x - x + xbottom1;

        Yu = (yright1 - yleft1)/2;
        Yv = (ytop1 - ybottom1)/2;
        Yuu = yright1 - y - y + yleft1;
        Yvv = ytop1 - y - y + ybottom1;

        Energy = w1*(Xu*Xu + Yu*Yu + Xv*Xv + Yv*Yv) +
          w2*(Xuu*Xuu + Yuu*Yuu + Xvv*Xvv + Yvv*Yvv);

        Xu = (xright2 - x)/2;
        Xuu = xright2 - xright1-xright1 + x;
        Yu = (yright2 - y)/2;
        Yuu = yright2 - yright1-yright1 + y;
        Energy += w1*(Xu*Xu + Yu*Yu) + w2*(Xuu*Xuu + Yuu*Yuu);

        Xu = (x - xleft2)/2;
        Xuu = x - xleft1-xleft1 + xleft2;
        Yu = (y - yleft2)/2;
        Yuu = y - yleft1-yleft1 + yleft2;
        Energy += w1*(Xu*Xu + Yu*Yu) + w2*(Xuu*Xuu + Yuu*Yuu);

        Xv = (xtop2 - x)/2;
        Xvv = xtop2 - xtop1-xtop1 + x;
        Yv = (ytop2 - y)/2;
        Yvv = ytop2 - ytop1-ytop1 + y;
        Energy += w1*(Xv*Xv + Yv*Yv) + w2*(Xvv*Xvv + Yvv*Yvv);

        Xv = (x - xbottom2)/2;
        Xvv = x - xbottom1-xbottom1 + xbottom2;
        Yv = (y - ybottom2)/2;
        Yvv = y - ybottom1-ybottom1 + ybottom2;
        Energy += w1*(Xv*Xv + Yv*Yv) + w2*(Xvv*Xvv + Yvv*Yvv);
      } ENDIF;
    }
    /* Energy rises w/radius, rises w/loss of continuity. */
    Energy += w3*dist;    /* w3 < 0; w1 > 0; w2 > 0; */
  ENDIF; ENDIF; ENDIF; ENDIF;
  retval = Energy; return(retval);
}
```

Figure 6.15: surface.C — Intensity Axis of Symmetry Program (part 6)

```
/* Use relaxation to find surface with minimal energy,
 * where energy is calculated by SurfEnergy().
 * See Tim Cullip's thesis for details.
 */
void converge_surface(simd_2_int X&, simd_2_int Y&,
       simd_2_int Z&, int Num, simd_2_int *D[],
       simd_2_int& F, simd_2gCsWrFlcL_bool_arr& R,
       GREYTYPE min, GREYTYPE max, simd_21L_int_arr& r)
{
  simd_2gCsWrFlcL_mach *m = R.m_2gCsWrFlcL();
  simd_2gCsWrFlcL_int x(m,GREYLEN), y(m,GREYLEN), z(m,GREYLEN);
  simd_2_int DX(m, GREYLEN), DY(m, GREYLEN);
  simd_2_int pChange(m, 32);
  simd_2_int Energy(m, 32), MinEnergy(m, 32);
  simd_2gCsWrFlcL_bool tmp(m);
  int Iteration;
  int Change;
  int SecondPass = Num;

  if (Num == 0) RadiusRidge(min, max, r, R);
  fflush(stdout);
  /* Flag indicates this point or a neighbor point has moved
   * in previous iteration, so this point should be considered
   * in this iteration.  After a point and its neighbors stop,
   * it stays stopped.  Set for self and neighbors if move.
   * This improves efficiency in original sequential program,
   * but is probably not needed in parallel versions. */
  F = 1;
  Iteration = 0;
  ZeroFlag = FALSE;
  /* Each iteration: for each pixel, check to see if its or
   * its neighbors' location is better place.  During 1st pass,
   * just move to higher radius number until hit a ridge.
   * 2nd pass: try to balance seeking high radius vs. keeping
   * local continuity. */
  while (Iteration < MaxIteration) {
    Change = 0;
    pChange = 0;
    Iteration++;
    index = 0;
    DX = DY = 0;
    x = X;
    y = Y;
    z = Z;
```

Figure 6.16: surface.C — Intensity Axis of Symmetry Program (part 7)

```
    /* Check energy at own and neighbor's location; if any
     * neighbor is better, remember its direction and move
     * in that direction. */
    IF (tmp.fetch(R, z, simd_label_2(x,y).pe_number()) == 0)
      /* Check my energy at my location */
      MinEnergy = SurfEnergy(x,y,0,D,F,SecondPass, m);
      Energy = SurfEnergy(x+1,y,1,D,F,SecondPass, m);
      /* Now check my energy at all 8 neighboring locations. */
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX=  1; DY=  0;} ENDIF;
      Energy = SurfEnergy(x-1,y,2,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX= -1; DY=  0;} ENDIF;
      Energy = SurfEnergy(x,y+1,3,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX=  0; DY=  1;} ENDIF;
      Energy = SurfEnergy(x,y-1,4,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX=  0; DY= -1;} ENDIF;
      Energy = SurfEnergy(x+1,y+1,5,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX=  1; DY=  1;} ENDIF;
      Energy = SurfEnergy(x-1,y+1,6,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX= -1; DY=  1;} ENDIF;
      Energy = SurfEnergy(x-1,y-1,7,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX= -1; DY= -1;} ENDIF;
      Energy = SurfEnergy(x+1,y-1,8,D,F,SecondPass, m);
      IF (Energy < MinEnergy)
        {MinEnergy = Energy; DX=  1; DY= -1;} ENDIF;
    } ENDIF;
    /* Move to better neighbor. */
    IF ((DX != 0) || (DY != 0)) {
      X += DX;
      Y += DY;
      F = 1;
      pChange++;
    } ELSE { F = 0; } ENDIF;
  }
  ZeroFlag = TRUE;
}

simd_int abs(const simd_int& val)
{ simd_int tmp=val;
  IF (tmp < 0) tmp = -tmp; ENDIF;  return(tmp);
}
```

Figure 6.17: surface.C — Intensity Axis of Symmetry Program (part 8)

may be **sun2**, **sun3**, **sun4**, **vax**, or **pxpl4gp** (for the PxPl4 Graphics Processor front-end). Also, include this compiler option: `-I${PORTA_SIMD}/include`. For example, the following command compiles the file `prog.c` for execution on a Connection Machine with a Sun-4 host. It uses the GNU C++ compiler, "g++".

```
g++ -I${PORTA_SIMD}/include -Dcm -Dsun4 -c prog.c
```

When linking the object files to generate an executable, specify linkage with a Porta-SIMD library using the `-l` option. Use library `simd_ARCHID`, where `ARCHID` is an architecture identifier. If the null architecture identifier is used, containing no optional features, omit the underscore and simply use library `simd`. Include this option in the link command: `-L${PORTA_SIMD}/lib/`*simd_hw*/*host_hw*/*compiler_name*. Here, *simd_hw* and *host_hw* are the symbols defined with `-D` in the compile command discussed in the preceding paragraph, and *compiler_name* names the C++ compiler being used. These symbols are needed only to manage the various platforms supported. If only one version of Porta-SIMD were installed, and it were installed in a standard system library directory, this `-L` link command option would not be needed at all. The implementation for a particular SIMD machine may require additional options on the link command, or even a special set of linkage commands. For example, linking for the CM-2 requires the options `-lparis` and `-lm`, and linking for PxPl4 requires two special commands. The commands below link `prog.o`, creating an executable program `prog` which runs on a Connection Machine with a Sun-4 host. The program has a target architecture identifier of `2aCG`.

```
setenv LIBDIR ${PORTA_SIMD}/lib/cm/sun4
CC -L${LIBDIR} -o prog prog.o -lsimd_2aCG -lparis -lm
```

6.2.7 Portability of Implementation

As already discussed in section 6.2.2, the prototype implementation of Porta-SIMD has been structured and written to maintain a clearly defined interface between machine-independent and machine-dependent parts of the program. All machine-dependent code is in the implementations of the virtual machine classes and certain memory management classes. In practice, even most of the code in these machine-dependent classes is shared by all versions. The first port to the CM-2, mentioned in section 6.2.1, took only five days, including the time required to learn Paris. The implementation has grown substantially since then, so there is more code to port, but it remains highly portable. Comparable figures are not available for the other ports, because they were done incrementally while other development work proceeded in parallel.

I estimate that porting Porta-SIMD to a new SIMD architecture requires writing a code fragment within the body of about 75 class methods. At least 50 of these code fragments require no more than five lines of code each in the existing ports. In existing ports, a handful of the fragments take 30-60 lines, and the remainder between five and 30 lines.

The use of C++ has contributed substantially to implementation portability. The AT&T and GNU C++ compilers are both highly portable and have both been widely ported.

The unique portability of Porta-SIMD's implementation is an attraction to the BLITZEN team, a member of which is considering porting it to the BLITZEN prototype.

6.3 Performance

This section evaluates from four perspectives the performance of potential and actual implementations of optimally portable languages. The first issue is how well some real SIMD architectures implement their equivalent architectures as defined by the taxonomy of chapter 3. Another way of stating this is, "what are some typical constants in some actual constant-operation simulations of equivalent architectures?" The second issue concerns the efficiency a high-quality compiler for an optimally portable language can be expected to achieve. In particular, can an optimally portable language be implemented as efficiently as existing high-level SIMD languages?

The final two issues study the efficiency of the prototype Porta-SIMD implementation. One examines the overhead in parallel operations and memory use. The other compares overall execution times of Porta-SIMD programs with versions of the same programs written in languages native to the SIMD machine.

6.3.1 Constant-Bounded Simulation

Chapter 3 presents a SIMD taxonomy and uses it to classify some SIMD architectures. It is reasonable to ask what constant values are involved in each real architecture's constant-bounded simulation of the architecture it is classified as having. Remember that a computer's architecture is defined as its lowest level publicly documented programming interface. In the vast majority of cases, across both operations and architectures, the constant is very small (1–5).

For example, most architectures provide almost all the arithmetic and logical operations of C++'s operators as single operations. Of course, there are a few exceptions: on some architectures, the constant for shift operators is the word length; on Pixel-Planes, multiply and divide require two or three times the operand length, and floating point operations require that or a little more; on the Oldfield machine, virtually all operations are laborious. Enable management for flow control takes one or two operations for most architectures, and rarely over four or five.

Communication operations which are supported directly by native operations can in almost every case be performed in one or two operations. The constant involved in constant-bounded simulation of other communication operations equivalent to the native operations depends on the operations involved. Simulating wrap-around communication with plain adjacent communication, by folding the machine in each dimension, usually requires four or six adjacent communication operations and four condition tests. Simulating diagonal communication takes one adjacent communication operation for each dimension. Simulating locally controlled adjacent communication where there is a fixed number of neighbors usually requires one `IF` and one `send()` operation per neighbor. Simulating combining in this case usually takes one additional operation per neighbor. Section 3.3 shows for each architecture which operations are supported by the native architecture and which are provided by constant-bounded simulation. The former are named in the architecture's *most natural* classification, (the first classification shown in each subsection of section 3.3); the latter are named in the *canonical* classification (shown second).

A separate issue is the execution time required by each real SIMD computer to execute the operations in its architecture (as defined by its lowest level publicly documented interface). This varies very widely. Bit-serial PEs typically require one to three clock cycles per bit for most operations. Multiplication, division, and floating point operations typically use clock cycles proportional to the square of the number of bits. Simple adjacent communication usually takes about the same number of clock cycles as an integer add, but on the CM-2 it takes several times longer [Thinking M89]. The time required for local addressing varies widely: BLITZEN does limited local addressing at comparable speed to global addressing; the MP-1 does unlimited local addressing in two or three times the time required for global addressing; and the CM-2 takes different times for limited and unlimited local addressing. Reduce and scan operations take time proportional to the logarithm of the number of processors on all the machines classified which provide them. The time required for global communication varies significantly with the particular communication pattern, the collision resolution mechanism, and the number of PEs.

6.3.2 Achievable Efficiency

A compiler for an optimally portable SIMD language should be able to produce code which executes just as efficiently as that produced by compilers for other high-level SIMD languages which provide equivalent features but are not optimally portable. The semantic analysis phase of a compiler for an optimally portable language can parse all declarations of architectural assumptions, verify that they are met, and remove them. This means the compiler's code generation phase receives essentially the same input as if the language were not optimally portable, and therefore should generate equally efficient code.

Alternatively, the compiler for the optimally portable language could use the compiler for the other language as a code generator. I.e., after parsing the program and verifying that it is consistent with its declared target architecture, the compiler could translate the program to the other language for compilation by the other compiler. Since the languages provide equivalent features, this should not be difficult in most cases. It should certainly provide programs written in the optimally portable language the same execution efficiency as if they had been written in the other language.

The following analysis of what is required for a language to be optimally portable supports the claim that a compiler for an optimally portable language should be able to generate code which executes as efficiently as that produced by a compiler for any other SIMD language providing equivalent features. Consider the three requirements of an optimally portable language, defined in chapter 2 and discussed in sections 5.1 and 6.1, and their effects on execution efficiency of compiled programs. The first and third requirements are that a target architecture be specified by the program, and that this specification be enforced by the compiler. The architecture specification is of course parsed and used by the semantic analysis phase of the compiler, and need not be passed to the compiler's code generator. The semantic analysis phase can also do the enforcement, using methods very similar to the type checking this phase already performs. The target architecture specification and enforcement requirements can therefore be satisfied without affecting at all the compiler's code generator or the execution performance of generated code.

Even the prototype Porta-SIMD implementation does most enforcement of target architecture declarations at compile time. Doing it all at compile time would be assisted by modifying the Porta-SIMD language slightly, so virtual machines would not have architecture identifiers. This would not sacrifice optimal portability, because all parallel variables would still have architecture identifiers. Requiring virtual machines to have them also is redundant. In any language where some of the architecture specification must be enforced at run time, this checking could be omitted at the user's request (as C and other languages often omit array bounds, overflow, and similar run time checks) to eliminate any effect on execution efficiency.

The remaining requirement is to provide language features for all architectural features supported by the computer's architecture. There are three classes of such language features.

- There are features also provided by the other language. These can be implemented identically, providing identical performance in their execution.
- There are features not provided by the other language, but which are special cases of features provided by the other language. For example, C* provides global communication but not grid communication. [1] Such features may either be implemented the

[1] There is a set of macros which provides grid communication, but it uses an escape to C/Paris and is not

same way as the more general case provided by the other language, or in a more efficient way taking advantage of the known special case nature of the feature. In neither case does the optimally portable language implementation need to be less efficient than the other language implementation, in which the programmer would be forced to use the more general feature.

- There are features supported by the architecture but not provided by the other language. The other language therefore forces the programmer to code a simulation of the feature. The optimally portable language implementation should be able to generate code which is at least as efficient as that programmed simulation.

So none of the language feature differences required for optimal portability detracts from the execution efficiency of code generated by the compiler for the optimally portable language.

6.3.3 Parallel Overhead in Porta-SIMD Prototype

The use of C++ `#include` files and libraries to implement Porta-SIMD places some significant limits on the implementation's efficiency. All parallel operations and data allocations (including temporary variables) are implemented by methods in a library of classes. The C++ compiler does not make any assumptions about what these methods do, and generates code to call them in a fixed and naive sequence for each SIMD expression. It does none of its usual optimizations on parallel expressions. So the operations and data elements which a compiler could save by register allocation, common sub-expression elimination, loop-invariant code movement, strength reduction, and induction variable and array index simplifications are not saved for parallel code.

The prototype Porta-SIMD implementation therefore has many characteristics of a threaded interpreter. There is no interpretive loop or repeated reparsing like an ordinary interpreter would have. But there is also no overall knowledge of expressions and program structure used in the execution. The program is simply compiled into a series of calls to the routines which interpret each operation. Therefore, most C++ operators with parallel operands produce a temporary result; if this result is assigned into a parallel variable, the assignment copies the temporary result. The only C++ operators which do not produce temporary results for parallel operands are ++, --, and the assignment operators (e.g., =, +=). Consider this example.

```
simd_int a, b, c;
a = b + c;
```

Porta-SIMD uses one temporary `simd_int`, one "add" operation, and one "copy" operation. Only the "add" would be needed if Porta-SIMD were implemented with a compiler.

In more complicated expressions where a compiler would reuse temporary variables, C++ creates a new one wherever one is needed.

```
simd_int a, b, c, d, e;
a = b + c + d + e;
```

Here, Porta-SIMD creates three distinct temporary variables, one for each operator invocation, when none are really needed. In fact, the situation is actually a little worse than that. For each of these temporary variables, the operator method that performs the addition creates an additional variable to hold the result, copies the result to the original temporary variable, and deletes the additional variable. These extra temporaries do not overlap in

part of the language itself. Even this package does not provide diagonal adjacent communication or piped adjacent communication with copy.

time, so they only add one variable to the memory consumed by the program, but they do add overhead for their allocation and freeing and for the extra copy.

Some of these temporary variables can be saved, if certain class operator methods are rewritten to mark the values they create as temporaries, and if these and other operator methods free a temporary parallel variable when they use its value. This would decrease maximum memory use, which would help programs that would otherwise run out of memory. It would not decrease parallel execution overhead, although one approach to doing that would be to avoid the final copy by using the temporary for the real variable and freeing the storage used by the real value instead of the temporary. But the sequential operations needed to do this correctly might easily take more time than they saved.

6.3.4 Overall Performance of Porta-SIMD Prototype

A limited amount of measurement has been performed to quantify the performance of Porta-SIMD programs with the prototype Porta-SIMD implementation. The method used is to compare *comparable programs* written in Porta-SIMD and another SIMD language. A program in another language is defined to be *comparable* to a Porta-SIMD program if it (a) uses the same algorithm, and (b) uses C or C++ (or something very similar) to express all sequential computations. It may use any available tool to express SIMD computations. Performance is stated as the ratio of the performance of a Porta-SIMD program and a comparable non-Porta-SIMD program.

The goal is to measure the performance of the Porta-SIMD implementation, not that of the algorithm or the C++ compiler. These must remain constant.

Measuring the overall performance of comparable programs gives a very complete performance measure, which includes sequential overhead in the Porta-SIMD implementation. However, it is imprecise because the definition of "comparable program" is necessarily imprecise. The definition was chosen to factor out (as much as possible) the performance of the C++ (or C) compiler used and the program's algorithm. Still, differences in programmer skill, time, and goals may yield programs with different performance in any language. Even if the same programmer writes both programs, has the same goals, and uses the same amount of time, the programmer's skill may still vary: he may be more familiar with one language than the other, or may be more familiar with the algorithm the second time he writes the program. In addition, lower-level tools than Porta-SIMD may yield superior performance at the expense of taking more programmer time.

The most reliable measurement, if least favorable for Porta-SIMD, may be to take stable production programs written without Porta-SIMD and rewrite them using Porta-SIMD. This assumes that such a program achieves approximately the best performance achievable, and that comparing it to a comparable Porta-SIMD program will therefore give a conservative estimate of Porta-SIMD's performance.

Greg Turk did exactly this with his Julia set program. It was a C program for PxPl4, with the parallel parts written with PxPl4's C-callable assembly-level "FB macros". It was hand-tuned for both speed and memory consumption. He measured its performance at 1500 iterations per second. There was sufficient memory to use 15-bit fixed point integers in the computation.

He translated the algorithm to Porta-SIMD, a significantly higher-level language. It was his first Porta-SIMD program, so he was less familiar with Porta-SIMD than with PxPl4's "FB macros". He measured the performance of the Porta-SIMD program on PxPl4 at 150 iterations per second, one tenth the speed of the comparable native PxPl4 program. There was sufficient memory to use only 10-bit fixed point integers. This means that despite some care in his coding, temporary values allocated by the prototype implementation of Porta-SIMD increased his program's memory use by about 50%.

Chapter 7

Recommendations

In the course of the research reported in this dissertation, I have developed some insights and opinions which are worth recording but which cannot necessarily be demonstrated or substantiated as correct. This chapter presents some of these insights and opinions in the form of recommendations for future work in various areas.

7.1 Prototype Porta-SIMD Implementation

The present implementation of Porta-SIMD has served primarily as a vehicle for language prototyping and exploration. It can continue to serve as a research tool in this role, particularly to explore greater use of operator syntax to express communication and local addressing as discussed in section 5.5. The prototype can also be extended to support floating-point data types and some additional architectural features. This will enhance its overall usability, which by attracting more users will make it a more valuable research tool.

It would be useful and straightforward to port Porta-SIMD to the MP-1, though the designs of MPL and the current Porta-SIMD implementation would necessitate some additional overhead not present in the other ports. The overhead is because parallel operations on the MP-1 can only be expressed in MPL, and the C++ translator can output only pure C. As a result, each C++ method in the Porta-SIMD implementation that normally does a parallel operation will instead call an MPL subroutine to do the operation for it. But I do not think this function call and return overhead per operation will detract significantly from the value of an MP-1 version of Porta-SIMD.

While Porta-SIMD has been and can continue to be a useful research tool, the current prototype implementation cannot be more than that. C++ `#include` files and libraries were the right tools for early and rapid language prototyping, but they are too limited to fully implement an optimally portable language. Only a compiler can implement all of Porta-SIMD's language features and architecture identifiers. And only a compiler can provide performance comparable to other compiled SIMD languages.

7.2 Porta-SIMD Language Design

Porta-SIMD meets its design goal, which was to demonstrate the design of an optimally portable SIMD language without sacrificing power or flexibility in other aspects of the language. As discussed in section 5.5, though, there are several important ways in which Porta-SIMD can be improved. While I think Porta-SIMD's design contributes some significant ideas to the field of SIMD language design, it is not yet mature enough to be the basis for a commercial compiler.

Porta-SIMD's design was influenced significantly by the tools being used to implement it. Those tools are not appropriate for a commercial or production quality implementation, so it is appropriate to reappraise the design decisions to which they contributed. That is the subject of the next section.

7.3 Improved Language Design

Deciding to develop a compiler to implement an optimally portable language would open up new design alternatives I did not explore fully in designing Porta-SIMD. The next few paragraphs discuss issues in optimally portable SIMD language design which I think it would be productive to explore further.

I would like to explore two ways of specifying target architectures that were not possible to implement with C++ `#include` files and libraries. Prepending `simd_ARCHID` to each basic data type name is far from ideal, so an obvious improvement is to make `simd` and the architecture identifier separate data attributes. Many existing SIMD languages use such an attribute to identify parallel data, so using another to identify its architectural assumptions is a natural extension. Another promising option is the scope-based target architecture specification described in section 5.1.1 on page 42. It is not yet clear which approach is better.

I have not yet seen a really satisfactory way of expressing all forms of inter-PE communication in any SIMD language. C* attempts to use only operators, and has some excellent ideas, but it provides no access to grid communication. MPL uses a combination of keywords and operators to express basic communication operations, and uses library routines for the rest. But some important variations are not available, including zero-edges instead of wrap and local addressing during communication. (The wrap/no-wrap option could be expressed by appending a character to the `xnet` keywords, which would be `w` or `0` to indicate wrap or zero edges.) Porta-SIMD's method-only mechanism is clearly not ideal. I think there is still room for improvement in this part of language design.

It is important to build on the best available sequential language. I strongly advocate C++ over C as a base language. C++ is more complex and less mature than C, so this may be difficult at first, but I believe that in the long run C++ will be a much better base language. The relative advantages of C++ and Fortran are probably less important than their different user populations, so there will be a need for both flavors of SIMD languages. Fortran-8X, or at least its data-parallel features, has already been recognized as the best Fortran base language.

It is important to support different sizes and shapes of SIMD variables, reflecting different kinds and sources of application data. Providing multiple virtual machines is one way to do this, but certainly not the only way.

Finally, compilers for SIMD languages need to do most of the same optimizations on parallel expressions and statements that they do on sequential code. There are opportunities for new optimizations unique to SIMD languages [Tuck87], but applying existing optimizations to parallel code is important and perhaps easier.

7.4 Host/PE-Array Interface

The interface between the host and the PE array needs some cooperative work involving compiler writers and architects. Existing SIMD systems implement this interface with different combinations of three basic elements: the host itself, a user-programmable sequential processor (the "controller"), and a microprogrammed sequencer (the "sequencer").

The CM-2, MP-1 and Pixel-Planes systems show typical uses of these elements. On the CM-2, user programs run on the host and send parallel operations to the sequencer; there is no controller. User programs on the MP-1 can run on both the host and the controller, but can only initiate parallel operations from the controller; there is no sequencer. Both PxPl4 and PxPl5 have all three elements. User programs run on the host and controller, like the MP-1, but the controller program sends instructions to the sequencer for parallel execution.

Using a controller can improve performance, but often makes programming less convenient. It is awkward and time consuming for the programmer to separate parallel programs into host and controller parts, as required on Pixel-Planes and when using MPL on the MP-1. The controller generally lacks some libraries and system calls present on the host, so parallel programs cannot necessarily run entirely on the controller. However, eliminating the controller, as the CM-2 does, has its own disadvantages. The most important is that existing hosts have insufficient bandwidth to the sequencer to keep it busy. The controller is normally specially designed to have much tighter coupling to the sequencer or PE array. Many programs also benefit from the host–controller division by overlapping execution of sequential user-interface operations on the host with parallel computation initiated from the controller, although this is at most a two-times speedup.

A sequencer implements more powerful primitive parallel operations than the bare PE array. This allows the controller (or host) to send fewer operations, reducing the sequential overhead of parallel operations and the bandwidth required to send them. The only reason I can see not to have a sequencer is if the performance increase it produces does not justify its hardware cost or design effort. PxPl5's sequencer is a single custom chip (including microcode store), so the hardware cost need not be large.

Several directions seem promising for improving the overall host to PE-array interface. One is developing smarter compilers capable of automatically dividing a program between the host and controller, relieving the programmer of this burden. This work is already under way at MasPar, where the MPF (MasPar Fortran) compiler generates code for both the host and the controller from a single user program. Another is improving the interface from the controller (or host) to the sequencer. The goal is to keep the sequencer (and therefore PE array) busy with minimum overhead on the controller (or host). This might be accomplished with more efficient hardware interfaces, or with higher-level sequencer primitives. The higher-level primitives could even be code fragments pulled from the program by the compiler.

Another direction is improving the host architecture. Existing SIMD systems use off-the-shelf UNIX-based systems as hosts. This has many advantages, including well-developed systems and applications software, wide variety of peripheral devices, and user familiarity. If a host-to-sequencer interface with adequate bandwidth and low host overhead could be developed without sacrificing the benefits of a standard host, it could eliminate the need to program the controller separately.

PxPl5 has multiple controllers and multiple sequencers, and host machines are beginning to have multiple processors. These can be powerful configurations, but there is a great deal we do not know about how to use them.

One useful feature of optimally portable languages for SIMD architects is that compilers for these languages should be able to collect very accurate statistics on the frequency of use of each architectural feature. This will be useful data in evaluating different host to PE-array interfaces to find configurations with the best performance and price/performance.

Chapter 8

Conclusion

The extraordinary architectural diversity of SIMD computers is too important to algorithm selection to completely hide from programmers. This dissertation presents *optimal portability*, a new concept for managing this architectural diversity. Optimal portability provides specific criteria for identifying the architectural features a programmer needs to see. It lets the programmer specify for each program the proper tradeoff between achieving broad portability and taking full advantage of a particular architecture. A new taxonomy is presented which facilitates the development of optimally portable languages. A new language, Porta-SIMD, is presented. It shows that optimal portability is an achievable goal for SIMD languages. A subset of Porta-SIMD has been implemented for the Connection Machine, two generations of Pixel-Planes, and sequential machines. Although the rapid prototyping implementation method does not maximize performance, analysis shows that optimally portable languages can be implemented as efficiently with respect to run time as existing high-level SIMD languages.

An optimally portable language for a set of architectures requires each program to specify its target architecture, i.e., its architectural assumptions and precise portability. An optimally portable language allows each program to use all the language features that are supported by the program's target architecture; but it does not allow the program to use any language feature not supported by that target architecture. An optimally portable language thereby lets the programmer judge the proper tradeoff between achieving broad portability and taking full advantage of a particular architecture. Existing languages foreclose this decision with predetermined architectural assumptions.

An optimally portable language requires an appropriate taxonomy of the set of architectures over which it is optimally portable. I present a taxonomy which uses constant-bounded simulation as its criterion for distinguishing architectures and their features. It is based on a precise definition of equivalence, defined by the requirements of optimal portability. This precise definition of equivalence brings to the surface more architectural dimensions than previous taxonomies used. As a result, each classification contains more descriptors and can be used to more nearly reconstruct the properties of the architecture it classifies than is possible with other taxonomies.

A subset of Porta-SIMD has been implemented to demonstrate the power and feasibility of optimally portable languages. This prototype implementation takes advantage of C++ classes and operator overloading to reduce the implementation effort. The prototype implementation runs on Pixel-Planes 4, the Pixel-Planes 5 simulator, the Connection Machine model 2, and sequential computers simulating SIMD architectures.

While this implementation method took good advantage of C++ to evolve a prototype Porta-SIMD implementation with a manageable effort, the limitations of C++ prevent the prototype from becoming a complete or highly efficient implementation. Any commercial-quality implementation of an an optimally portable language will require a real compiler. This will allow improvements in the syntax and semantics of the language, and make it possible to provide a complete and efficient implementation.

Although optimal portability has been applied here to SIMD architectures, it is potentially applicable to any diverse but related class of architectures, e.g., graphics engines and MIMD (Multiple-Instruction, Multiple-Data) architectures.

Appendix A

Note on Prototype Porta-SIMD `#include` Files

The appendix "Prototype Porta-SIMD **`#include`** Files" is too large to include in this technical report. It is included in the official dissertation, which is archived in the library of Duke University.

Copies of the official dissertation, including the appendix, are available for purchase from UMI Dissertation Services, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, USA; (800) 521-0600, (313) 761-4700.

Bibliography

[Abuhamdeh88] Zahi Abuhamdeh. The GAM II pyramid. In Mills [Mills88], pages 443–448.

[Active Mem87a] Active Memory Technology Limited, Irvine, CA. *DAP 500: APAL Language*, 1987.

[Active Mem87b] Active Memory Technology Limited, Irvine, CA. *DAP 500: Introduction to FORTRAN-PLUS Programming*, 1987.

[Active Mem88] Active Memory Technology, Inc., Irvine, CA. *AMT DAP Series Technical Overview*, 1988.

[AlbertKnLS88] 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).

[BallBolJMS62] J. R. Ball, R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, and D. H. Shaffer. On the use of the SOLOMON computer parallel-processing computer. In *Proceedings of the Fall Joint Computer Conference*, volume 22, pages 137–146. AFIPS, December 1962.

[BeetemDenW85] 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.

[BeetemDenW86] John Beetem, Monty Denneau, and Don Weingarten. GF11. *Journal of Statistical Physics*, 43(5/6), June 1986.

[BlaauwBroo90] Gerrit A. Blaauw and Frederick P. Brooks, Jr. *Computer Architecture*, volume 1. Addison-Wesley Publishing Company, Reading, MA, 1990. In press.

[Blelloch87] Guy Blelloch. Scans as primitive parallel operations. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 355–362, University Park, PA, August 1987. The Pennsylvania State University Press.

[BlevinsDaR87] Donald W. Blevins, Edward W. Davis, and John H. Reif. Processing element and custom chip architecture for the BLITZEN massively parallel processor. Technical Report TR87-22, Microelectronics Center of North Carolina, October 1987. Revised Edition, June 1988.

[BlevinsDHR90] Donald W. Blevins, Edward W. Davis, Robert A. Heaton, and John H. Reif. BLITZEN: A highly integrated massively parallel machine. *Journal of Parallel and Distributed Computing*, 8(2):150–160, February 90. Originally published in [Mills88, pp. 399-406].

[ChinPasBTK88] D. Chin, J. Passe, F. Bernard, H. Taylor, and S. Knight. The Princeton Engine: A real-time video system simulator. *ICCE*, May 1988.

[Collins90] Robert Collins. *CM++ Manual, Release 2.02.* Department of Computer Science, University of California Los Angeles, April 1990. Available by anonymous ftp from polaris.cognet.ucla.edu, in cm++-2.03.tar.Z.

[Dasgupta89] Subrata Dasgupta. *Computer Architecture: A Modern Synthesis*, volume 1. John Wiley & Sons, New York, 1989.

[DavisReif88] 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.

[Duff83] M. J. B. Duff, editor. *Computing Structures for Image Processing.* Academic Press, Inc., Orlando, FL 32887, 1983.

[EylesAuFGP87] 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.

[Feng77] T. Y. Feng. Parallel processors and processing. *ACM Computing Surveys*, 9(1), March 1977.

[Flynn66] Michael J. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[Fountain83] T. J. Fountain. A survey of bit-serial array processor circuits. In Duff [Duff83], chapter 1, pages 1–14.

[FuchsGHSA+85] 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).

[FuchsPEGG+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, 23(3):79–88, July 1989. (Proceedings of SIGGRAPH '89).

[FuchsPoult81] Henry Fuchs and John Poulton. Pixel-Planes: a VLSI-oriented design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

[GauchPizer88] John M. Gauch and Stephen M. Pizer. Image description via the multiresolution intensity axis of symmetry. Technical Report TR88-046, University of North Carolina at Chapel Hill, Department of Computer Science, September 1988. Also published in ICCV '88 Proceedings.

[Gerritsen83] F. A. Gerritsen. A comparison of the CLIP4, DAP, and MPP processor-array implementations. In Duff [Duff83], chapter 2, pages 15–30.

[Goodyear A83a] Goodyear Aerospace Corp. *MPP Main Control Language — MCL, GER-16672*, April 1983.

[Goodyear A83b] Goodyear Aerospace Corp. *MPP PE Array Language — PRL, GER-16655*, March 1983.

[GreerEyles88] Trey Greer and John Eyles. *Pixel-Planes 4 Programmer's Intro*, 1988.

[HametDorba88] L. Hamet and J. Dorband. A generic fine-grained parallel C. In Mills [Mills88], pages 625–628.

[Handler77] W. Händler. The impact of classification schemes on computer architecture. In *Proceedings of the 1977 International Conference on Parallel Processing*, 1977.

[HarbisonSt84] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[Hillis85] W. Daniel Hillis. *The Connection Machine.* MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, MA, 1985.

[Hord82] R. Michael Hord. *The Illiac IV: The First Supercomputer.* Computer Science Press, Inc., Rockville, MD, 1982.

[HwangBrigg84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

[Iliffe82] J. K. Iliffe. *Advanced Computer Design.* Prentice/Hall International, London, 1982.

[Jamieson87] Leah H. Jamieson. Features of parallel algorithms. In *Second International Conference on Supercomputing, Vol. 1*, pages 476–478. International Supercomputing Institute, Inc., 1987.

[Karp87] Alan H. Karp. Programming for parallelism. *Computer*, 20(5):43–56, May 1987.

[KernighanR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[Kohonen80] Teuvo Kohonen. *Content Addressable Memories.* Springer-Verlag, Berlin, 1980.

[KrikelisLe88a] A. Krikelis and R. M. Lea. An associative approach to computer vision. In Levialdi [Levialdi88], pages 75–95. Proceedings of the Eighth Workshop on Multicomputers, held in Rome, Italy on June 2-5, 1987.

[KrikelisLe88b] A. Krikelis and R. M. Lea. Low-level vision tasks using parallel string architectures. In *Parallel Processing for Computer Vision and Display*, January 1988.

[KrikelisLe88c] A. Krikelis and R. M. Lea. Performance of the ASP on the DARPA architecture benchmark. In Mills [Mills88], pages 483–486.

[Lea86a] R. M Lea. SCAPE: A single-chip array processing element for signal and image processing. *IEE Proceedings*, 133, part E(3):145–151, May 1986.

[Lea86b] R. M Lea. VLSI and WSI associative string processor for structured data processing. *IEE Proceedings*, 133, part E(3):153–162, May 1986.

[Lea88] R. M Lea. ASP: A cost-effective parallel microcomputer. *IEEE Micro*, 8(5):10–27, October 1988.

[Levialdi88] S. Levialdi, editor. *Multicomputer Vision*. Academic Press, Inc., San Diego, CA 92101, 1988. Proceedings of the Eighth Workshop on Multicomputers, held in Rome, Italy on June 2-5, 1987.

[Liebrick79] J. P. Liebrick. Burroughs super-computer scheduled for Japan. *Burroughs B Line*, page 1, 1979. Spring Edition.

[Lippman89] Stanley B. Lippman. *C++ Primer*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[MarescaLi88] Massimo Maresca and Hungwen Li. Toward connection autonomy of fine grain SIMD parallel architecture. In *Parallel Processing for Computer Vision and Display*, January 1988.

[MarescaLi89] Massimo Maresca and Hungwen Li. Connection autonomy in SIMD computers: A VLSI implementation. *Journal of Parallel and Distributed Computing*, 7(2):302–320, October 1989.

[MasPar Com90a] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, 1990.

[MasPar Com90b] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) User Guide*, 1990.

[Massachuse84] Massachusetts Computer Associates, Inc. *A FORTRAN Compiler for the Massively Parallel Processor*, February 1984. CADD-8402-2101.

[MiddletonT89] David Middleton and Sherry Tomboulian. Evaluating local indirect addressing in SIMD processors. ICASE Report 89-30, Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665-5225, May 1989.

[MillerStou88a] Russ Miller and Quentin R. Stout. An introduction to the portable parallel programming language Seymour. In *Proceedings of the IEEE Computer Society's Thirteenth Annual International Computer Software and Applications Conference*, 1988.

[MillerStou88b] Russ Miller and Quentin R. Stout. Portable parallel algorithms for geometric problems. In Mills [Mills88], pages 195–198.

[Mills88] Ronnie Mills, editor. *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation (Frontiers '88)*, Washington, DC, October 1988. Computer Society of the IEEE, NASA/Goddard Space Flight Center, IEEE National Capital Area Council, and George Mason University, IEEE Computer Society Press.

[NASA/Godda85a] *MPP Main Control Language — MCL*, September 1985. Revised March 1986. Prepared for NASA/Goddard Space Flight Center (Greenbelt, MD 20771) by Science Applications Research (Lanham, MD 20706). Based on [Goodyear A83a].

[NASA/Godda85b] *MPP PE Array Language — Pearl*, September 1985. Prepared for NASA/Goddard Space Flight Center (Greenbelt, MD 20771) by Science Applications Research (Lanham, MD 20706), under Contract NAS 5-28200, Task Assignment 177. Based on [Goodyear A83b].

[Odonnell88] John T. O'Donnell. MPP implementation of abstract data parallel architectures for declarative programming languages. In Mills [Mills88], pages 629–636.

[OldfielWWB88] 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.

[ParkinsoHM88] 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.

[Perrott75] R. H. Perrott. A language for array and vector processors. *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, October 1975.

[Potter85] J. L. Potter, editor. *The Massively Parallel Processor*. MIT Press Series in Scientific Computation. The MIT Press, Cambridge, MA, 1985.

[PreparataV81] 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.

[RiceSeidmW88] M. D. Rice, S. B. Seidman, and P. Y. Wang. A formal model for SIMD computation. In Mills [Mills88], pages 601–607.

[RiceSeidmW90] Michael D. Rice, Stephen B. Seidman, and Pearl Y. Wang. The specification of data parallel algorithms. *Journal of Parallel and Distributed Computing*, 8(2):191–195, February 90.

[RosenberBH88] Jonathan B. Rosenberg, Jonathan D. Becher, and Nigel F. Hooke. Vectorization enables full-scale simulation of massively parallel (SIMD) architectures. In *Third International Conference on Supercomputing, Vol. 3*, pages 305–312. International Supercomputing Institute, Inc., 1988.

[RosenbergD88] Jonathan B. Rosenberg and Edward W. Davis. BLITZ: Blitzen's μ-code assembly language, design document, version 1.0. Technical Report TR88-14, Microelectronics Center of North Carolina, March 1988.

[RoseSteele87] 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.

[Sabot87] Gary W. Sabot. The paralation model as a basis for parallel programming languages. Technical Report PL87-3, Thinking Machines Corporation, April 1987.

[Sabot88a] Gary W. Sabot. Paralation Lisp reference manual. Technical Report PL87-11, Thinking Machines Corporation, May 1988.

[Sabot88b] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, Cambridge, MA, 1988.

[SchaefeHBV87] D. H. Schaefer, P. Ho, J. Boyd, and C. Vallehos. The GAM pyramid. In Leonard Uhr, editor, *Parallel Computer Vision*, pages 15–42. Academic Press, Inc., Orlando, FL 32887, 1987.

[Schaefer90] David H. Schaefer. Special issue on massively parallel computation: Guest editor's introduction. *Journal of Parallel and Distributed Computing*, 8(2):101, February 90.

[Siegel85] Howard Jay Siegel, editor. *Interconnection Networks for Large-Scale Parallel Processing.* Lexington Books, Lexington, MA, 1985.

[SlotnickBM62] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The SOLOMON computer. In *Proceedings of the Fall Joint Computer Conference*, volume 22, pages 97–107. AFIPS, December 1962.

[Snyder88] Lawrence Snyder. A taxonomy of synchronous parallel machines. In Fayé A. Briggs, editor, *Proceedings of the 1988 International Conference on Parallel Processing*, volume 1 Architecture, pages 281–285, University Park, PA, August 1988. The Pennsylvania State University Press.

[SteeleHill86] 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.

[Stroustrup86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[Stroustrup89] Bjarne Stroustrup. *AT&T C++ Language System Product Reference Manual*, May 1989. Select Code 307-146.

[Taylor88] Herb Taylor, 1988. Personal communication.

[Technical87] Technical Committee X3J3–Fortran, Accredited Standards Committee X3–Information Processing Systems. *X3.9-198x: Draft Proposed Revised American National Standard Programming Language Fortran (Version 104).* American National Standards Institute, April 1987.

[Thinking M87a] Thinking Machines Corporation, Cambridge, MA. *Connection Machine Parallel Instruction Set (Paris): The C Interface (Version 4.0)*, 1987.

[Thinking M87b] Connection Machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Corporation, April 1987. Revised version 5.1, May 1989.

[Thinking M88] Thinking Machines Corporation, Cambridge, MA. *Supplement to *Lisp Reference Manual*, June 1988. Version 5.0A Field Test.

[Thinking M89] Thinking Machines Corporation, Cambridge, MA. *Paris Reference Manual, Version 5.2*, October 1989.

[TombouliMW85] Sherry J. Tomboulian, Mary Mace, and Robert A. Wagner. Language report and description for BVL-0. Unpublished paper, April 1985.

[Tuck85] Russ Tuck. BVL-0: A language for SIMD computing. Unpublished paper, October 1985.

[Tuck87] Russ Tuck. Issues in the design of an optimizing code generator for BVL-0. Master's thesis, Duke University, Durham, NC, 1987.

[Tuck88] Russ Tuck. An optimally portable SIMD programming language. In Mills [Mills88], pages 617–624. Also available as University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR88-048.

[Tuck89] Russ Tuck. Porta-SIMD user's manual. Technical Report TR89-006, University of North Carolina at Chapel Hill, Department of Computer Science, January 1989. Revised March 1989.

[UNC MSL88] Microelectronic Systems Laboratory, Department of Computer Science, University of North Carolina at Chapel Hill. *Pixel-Planes 5 System Documentation*, 1988.

[Unger58] S. H. Unger. A computer oriented toward spatial problems. *Proceedings of the IRE*, 46(10):1744–1750, October 1958.

[Wagner81] Robert A. Wagner. A programmer's view of the Boolean Vector Machine, model-2. Technical Report CS-1981-8, Department of Computer Science, Duke University, Durham, NC 27706, October 1981.

[Wagner83] 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.

[Wilson88] Stephen S. Wilson. One dimensional SIMD architectures—the AIS-5000. In Levialdi [Levialdi88], pages 131–149. Proceedings of the Eighth Workshop on Multicomputers, held in Rome, Italy on June 2-5, 1987.

[Wirbel89] Loring Wirbel. IBM's parallel efforts. *Electronic Engineering Times*, page 32, May 15, 1989.

[WuFeng84] Chuan-lin Wu and Tse-yun Feng, editors. *Tutorial: Interconnection Networks for Parallel and Distributed Processing.* IEEE Computer Society Press, Silver Spring, MD 20910, 1984.