Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language/Compiler

Mark A. Nichols, *Member, IEEE*, Howard Jay Siegel, *Fellow, IEEE*, and Henry G. Dietz, *Member, IEEE*

***Abstract*—Features of an explicitly parallel programming language targeted for reconfigurable parallel processing systems, where the machine's N processing elements (PE's) are capable of operating in both the SIMD and SPMD modes of parallelism, are described. The SPMD (Single Program–Multiple Data) mode of parallelism is a subset of the MIMD mode where all processors execute the same program. By providing all aspects of the language with an SIMD mode version and an SPMD mode version that are syntactically and semantically equivalent, the language facilitates experimentation with and exploitation of hybrid SIMD/SPMD machines. Language constructs (and their implementations) for data management, data-dependent control-flow, and PE-address dependent control-flow are presented. These constructs are based on experience gained from programming a parallel machine prototype, and are being incorporated into a compiler under development. Much of the research presented is applicable to general SIMD machines and MIMD machines.**

***Index Terms*— Compilers, fault tolerance, languages, mixed-mode parallelism, parallel processing, PASM, reconfigurable machines, SIMD, SPMD.**

I. Introduction

TWO approaches to parallel computation are the SIMD and MIMD modes of parallelism. In the synchronous *SIMD* (Single Instruction—Multiple Data) [18] mode of parallelism, the processors are broadcast all of the instructions they are to execute and the processors operate in a lock-step fashion. In the asynchronous *MIMD* (Multiple Instruction—Multiple Data) [18] mode of parallelism, each processor functions independently using its own program. In the *SPMD* (Single Program—Multiple Data) [14], [15] mode, all processors function independently using their own program counters and all execute the same program. Because the processors will, in general, be operating independently on different data sets, they may be taking different control paths through the program.

On parallel systems capable of operating only in a single mode of parallelism, it is often impossible to fully exploit the parallelism inherent in many application programs. This is because different parts of an application program might map more closely to different modes of parallelism [19], [23]. For instance, most parallel systems designed to exploit data parallelism operate solely in the SIMD mode of parallelism. Because many data-parallel applications require a significant number of data-dependent conditionals, SIMD mode is unnecessarily restrictive. These types of applications are usually better served when using the SPMD mode of parallelism. Several parallel machines have been built that are capable of operating in both the SIMD and SPMD modes of parallelism. Both PASM [9], [17], [51], [52] and TRAC [30], [46] are hybrid SIMD/MIMD machines, and OPSILA [2], [3], [16] is a hybrid SIMD/SPMD machine.

The research presented here is applicable to SIMD, SPMD, and combined SIMD/SPMD operation. Commercial SIMD machines like the Connection Machine [21], [54], DAP [22], MasPar [8], and MPP [5], [6] have shown that developing parallel languages for programming and studying the behavior of such machines is important. While commercial MIMD machines, such as the Butterfly [13], iPSC Hypercube [35], and NCube system [20], are not restricted to SPMD mode, SPMD mode has wide applicability and is simpler to use because only one program and its interaction with copies of itself needs to be debugged (as opposed to having distinct interacting programs on different processors). For joint SIMD/SPMD machines such as PASM and OPSILA, because there exist tradeoffs between the two modes [7], [47], one can take advantage of both modes to attempt to improve performance.

In addition to allowing the use of the more suitable mode when executing a task, being able to utilize two modes of parallelism within parallel programs provides more flexibility with respect to fault tolerance. Given a task that executes more effectively in SIMD mode than in SPMD mode, if a processor becomes faulty, the degraded system may be more effective in SPMD mode rather than in SIMD. Reasons for this may include that the needed redistribution of work among the processors will require irregular inter-processor data transfers and that the addressing requirements of the degraded system may no longer be uniform among the processors. Thus, being able to execute the same program both in SIMD mode or SPMD mode is needed to facilitate such a fault tolerance approach. The language presented here will support this.

The Explicit Language for Parallelism (*ELP*) is an explicitly parallel language being designed for programming parallel machines where the user explicitly indicates the parallelism to be employed. It will include constructs for both SIMD and MIMD parallelism. Thus, the full ELP language will be adaptable to SIMD machines, MIMD machines, and machines

Manuscript received October, 15, 1990; revised September 1, 1991. This work was supported by the Office of Naval Research under Grant N00 014-90-J-1483, by the National Science Foundation under Grant CDA-9015696, and by the Naval Ocean Systems Center under the High Performance Computing Block, ONT. A preliminary version of portions of this material was presented at Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation.

M. A. Nichols is with NCR, San Diego, CA 92127-1806.

H. J. Siegel and H. G. Dietz are with the Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907-1285.

IEEE Log Number 9204628.

capable of operating in both the SIMD and MIMD modes. An ELP application program is able to perform computations that use the SIMD and MIMD parallelism modes in an interleaved fashion.

The first target architecture for ELP is the *PASM* (Partitionable SIMD/MIMD) parallel processing system. ELP is currently being developed on a small-scale 30-processor prototype of PASM (with 16 processors in the computational engine). PASM is a design for a large-scale reconfigurable system with distributed memory that supports *mixed-mode* parallelism (i.e., its processing elements (PE's) can dynamically switch between the SIMD and MIMD modes at instruction level granularity with negligible overhead). ELP uses *explicit* specification of parallelism so that programmers can conduct experiments to study the use of SIMD, MIMD, and mixed-mode parallelism. Thus, the goal of ELP is to facilitate the programming of and experimentation with mixed-mode parallel processing systems by supporting explicit control of such systems.

Many parallel algorithm studies have been performed on the PASM prototype. Initially, these studies used assembly language, e.g., [9], [17]. Currently, as a temporary measure, a combination of a C language compiler and AWK scripts (for pre- and post-processing) is used. Even though this permits the high-level specification of parallel programs, it is very inefficient and inflexible. These practical experiences in mixed-mode parallel programming and discussions with members of the PASM user community provided the motivation for the development of the ELP language/compiler.

This paper focuses on aspects of the joint SIMD/SPMD portion of the ELP language/compiler. Also, the compilation/linking/loading process is described. In particular, a variety of inter-related data management, data-dependent control-flow, and PE-address dependent control-flow issues are examined. Their specification and implementation are discussed. Allowing the user to explicitly program mixed-mode SIMD/SPMD operation introduces problems not found if just a single mode were used. The language concepts presented may be adapted for use in most distributed-memory architectures capable of SIMD, SPMD, or SIMD/SPMD operation.

Section II addresses related work and Section III describes the parallel machine model. Joint SIMD/SPMD data management in ELP is presented in Section IV. Joint SIMD/SPMD data-dependent control-flow constructs and joint SIMD/SPMD PE-address dependent control-flow constructs are covered in Sections V and VI, respectively. In Section VII, SIMD/SPMD execution mode specification is discussed. Finally, the compilation, linking, and loading process for ELP is overviewed in Section VIII.

II. Related Work

Many parallel languages have been developed or proposed that are based on either the SPMD mode of parallelism or the SIMD mode of parallelism. SPMD-based parallel languages include the Force [24], [25], the Uniform System [53], and the EPEX/FORTRAN System [14], [15]. SIMD-based parallel languages include CFD [1], Glypnir [29], Actus [37]–[39], DAP Fortran [22], MasPar C and Fortran [11], Parallel-C [28], Parallel Pascal [42], extensions to Ada [12], C* [44], and DAPL [43], [55].

Existing parallel languages capable of supporting mixed-mode operation include CSL, Hellena, and BLAZE. While CSL and Hellena are targeted for the TRAC and OPSILA machines, respectively, BLAZE is untargeted.

CSL (Computation Structures Language) [10] is a PASCAL-like explicitly parallel job control language, targeted to TRAC, that supports the high-level specification and scheduling of SIMD tasks (specified via an "`EXECUTE`" statement) and MIMD tasks (specified via a "`COBEGIN . . . COEND`" statement). Dynamic switching between SIMD mode and MIMD mode is performed at the task level.

Hellena [3] is an explicitly parallel preprocessed version of PASCAL, targeted to OPSILA, that supports dynamic mode-switching at the instruction level. SIMD mode is the default mode of execution, and the encapsulation "`spmd . . . end_spmd`" can be used to temporarily switch to SPMD mode.

Unlike the previous two languages, BLAZE [31] (targeted for machines capable of SIMD and/or SPMD operation) handles parallelism in an implicit fashion. Specifically, BLAZE supports language constructs that provide multiple levels of parallelism that permit a compiler to extract the types of parallelism appropriate for a given architecture, while discarding the remainder.

A mixed-mode language is *uniform* with respect to each mode it supports if *all* of the language's constructs, operations, statements, etc., have interpretations within each of the modes with no difference in syntax or semantics [40]. The first parallel language proposed that supported this concept was XPC. XPC (eXplicitly Parallel C) [40], [41] is a C-like language that provides explicit specification and management of both the synchronous and asynchronous paradigms of parallel programming. XPC does not assume a particular execution model (e.g., SIMD machine, MIMD machine, SIMD/MIMD machine).

The syntax of ELP is based on C [26] and has been extended with parallel constructs and specifiers (building on XPC and Parallel-C). The SIMD and SPMD modes of parallelism are supported by a full native-code compiler (under development), targeted to PASM, that permits these modes to be switched dynamically at instruction level granularity. (ELP will be extended to include full MIMD capability.) ELP is a mixed-mode language that is uniform with respect to the SIMD and SPMD modes of parallelism. This feature allows data-parallel algorithms to be coded in a mode-independent manner, after which execution mode specifiers (see Section VII) easily can be added and changed when the program is to be compiled. It also helps support mixed-mode experimentation and reconfiguration for fault tolerance. The mapping of data to memories to support this is discussed in [33].

A significant difference between ELP and both CSL and Hellena is that ELP is uniform with respect to its modes of operation, while CSL and Hellena are highly nonuniform with respect to their modes of operation. Additional differences among ELP, CSL, and Hellena are: 1) ELP performs mode-switching at the instruction level whereas CSL performs it at the task level and 2) ELP is supported by a full native-code compiler whereas Hellena is preprocessed. Because BLAZE is an implicitly parallel language, comparisons with ELP's explicit constructs are not appropriate.

III. Parallel Machine Model

This section defines the parallel machine model required for *joint* SIMD/SPMD operation in ELP. As discussed, the results proposed in this paper can also be applied to single-mode machines capable of SIMD or SPMD operation. After describing the general model, relevant features of a machine representative of this model are provided.

It is assumed that the machine can operate in both the SIMD and SPMD modes of parallelism and that switching between these modes can be accomplished at the instruction level. An additional requirement is that memory be distributed among all processors. In SIMD mode, it is assumed that there is a control unit in charge of broadcasting instructions to the processors and that it is also capable of performing any scalar computations. In SPMD mode, it is assumed that the processors use their own program counters to execute an identical program located within their local memories. Special hardware must exist for: 1) broadcasting data values from the control unit to all processors, 2) sending a data value from an arbitrary processor to the control unit, and 3) permitting the control unit to determine whether or not all processors possess a particular value. The machine must also support efficient SPMD barrier synchronization (i.e., forcing all processors to synchronize at a specific program location). Lastly, it is assumed that there are $N = 2^n$ *PE's* (Processing Elements—processor/memory pairs) addressed 0 to $N - 1$ and that each PE can access its unique address (or number) at execution-time.

The following material overviews the relevant portions of the architecture of the PASM parallel processing system as an example of a machine in the class described by the above model. The set of $N = 2^n$ PE's within PASM can be partitioned to form one or more independent *submachines* of various sizes (under certain constraints [52]). Each submachine has its own "virtual" SIMD *CU* (Control Unit). In PASM, the CU has access to special hardware for determining whether or not all PE's possess a particular value (e.g., an "if-any" statement) [45]. Additionally, each PE has execution-time access to a memory-mapped location containing its unique address (or number).

Each CU includes an *FU* (Fetch Unit). In SIMD mode, the CU CPU initiates parallel computation by sending blocks of SIMD code from the FU memory (which contains the SIMD code) to the FU queue. Once in the FU queue, each SIMD instruction is broadcast to all PE's. While the FU is enqueuing and broadcasting SIMD instructions to the PE's and the PE's are executing instructions, the CU CPU can be performing its own computations—this phenomenon is called *CU/PE overlap* [27]. The FU queue can also be used to broadcast data values from the CU to all PE's. Many pure SIMD machines have a CU/PE overlap capability (e.g., MPP [6], Illiac IV [4]).

In SPMD mode, the PE's operate independently using their own program counters and execute an identical program located within their local memories. By performing a barrier synchronization, data values can be broadcast from the CU to all PE's via the FU queue. Additionally, data values can be transferred from a specific PE to the CU (and vice versa) via a specialized hardware bus.

Switching between SIMD mode and MIMD mode on PASM is handled by dividing the PEs' logical address space into an MIMD address space, where the PE's access their own local memory, and an SIMD address space, where the PE memory requests are satisfied by the FU broadcasting SIMD instructions. Therefore, switching a submachine from SIMD to MIMD is implemented by broadcasting to the PE's a branch to MIMD space, while switching from MIMD to SIMD is implemented by all PE's independently branching from MIMD to SIMD space. Recall that SPMD mode is just a special case of MIMD mode. As can be seen, changing execution modes in PASM is performed solely by changing the source of control for the PE's (i.e., changing the PEs' source of instructions to execute) and everything else (memory, registers, processor state, etc.) is unaffected when changing modes. At any point in time, all PE's in a given submachine must be in the same execution mode.

Given an ELP source file, the ELP compiler generates three distinct assembly output files. One is for the CU, one is for the FU, and one is for the PE's. The CU file contains scalar instructions and data, and commands for broadcasting instructions to the PE's. The FU file contains parallel instructions to be broadcast to the PE's. Lastly, the PE file contains instructions to be executed in SPMD mode and data for both SIMD and SPMD operation. Further information can be found in Section VIII.

IV. Joint SIMD/SPMD Data Management

This section describes data management issues for a distributed-memory program that uses both SIMD mode and SPMD mode and switches between the two modes one or more times at instruction level granularity. A discussion of these issues for programs that execute entirely in either SIMD mode or SPMD mode is provided at the end of this section.

A. Variable Classes

All variables defined in ELP have a *variable class* associated with them. A variable defined to be of class *mono* always has a single value with respect to all PE's, independent of execution mode (i.e., a mono variable is scalar-valued); whereas a variable defined to be of class *poly* can have one or more values with respect to all PE's, independent of execution mode (i.e., a poly variable is vector-valued). (Execution mode specification in ELP is statically scoped and uses the keywords `simd` and `spmd`—see Section VII.) Each mono variable (whether a global, local, or parameter) has storage allocated for it on the CU and all PE's. If a mono variable is referenced while in SIMD mode, its CU storage is active. If a mono variable is referenced while in SPMD mode, its PE storage is active and all PE copies of the mono variable will have the same value (guaranteed by the compiler). More precisely, in SPMD mode mono variables are guaranteed to possess the same value across all PE's at any point *in an ELP program,* not necessarily at any point in time. For variables defined to be poly, each PE has its own copy with its own value, independent of execution mode.

The use of mono variables in an ELP program has different implementations depending on the execution mode employed. In SIMD mode, mono variables and operations on mono variables permit work to be done on the CU, and they permit CU/PE overlap to be explicitly specified. This, in turn, allows the user to experiment with load balancing between the CU and the PE's. In SPMD mode, mono variables can be used to force `if`, `while`, `do`, and `for` statements on different PE's to execute in the same fashion on all PEs; for example, mono variables could be used as the index variable and as the common upper bound for a `for` loop with all PE's. All PE's must execute the same instructions, but not necessarily at the same time as in SIMD mode. Mono variables also permit other SPMD operations to be performed in an identical fashion across all PE's, such as having each PE access the same element of an array. An example of declaring a mono integer variable `x` is: "`mono int x;`".

In a nonpartitionable machine, a poly variable declaration invokes a variable on each PE. If the target machine is partitionable, the *extent of parallelism* is the number of PE's specified by the user to be in the submachine. On a partitionable machine, if an ELP program does not use any selector statements (defined in Section VI), it can be executed with many different extents of parallelism and a particular extent does not need to be specified until load-time. Specifically, the user is required to specify the submachine size (extent of parallelism) when loading an ELP program. ELP programs that employ selector statements must be recompiled whenever a different extent of parallelism is desired; this is discussed further in Section VI. An example of declaring a poly array variable `y` is: "`poly char y[10];`". This declaration would allocate an array of ten characters on each PE within the extent of parallelism specified at load-time.

The keywords mono and poly are used in both C* (targeted solely for SIMD mode) and XPC (a machine-independent language). ELP's interpretation of these keywords differs from these languages due to ELP's explicit nature and target of distributed-memory SIMD/SPMD machine architectures.

B. Assignment Statements

Code generation for an assignment statement in ELP is dependent on the execution mode, the variable class of the *Lvalue* (left side of the statement), and the variable class of the *Rvalue* (right side of the statement). Tables I and II describe the code generated for assignment statements in SIMD mode and SPMD mode, respectively.

ELP assignment statements in SIMD mode are similar to assignment statements proposed in the SIMD portion of the Parallel-C language [28]. In SIMD mode, when the Lvalue and Rvalue are both mono, a standard assignment is performed within the CU. Similarly, when the Lvalue and Rvalue are both poly, a standard assignment instruction is broadcast to (and subsequently performed within) each PE. In SIMD mode, when the Lvalue is poly and the Rvalue is mono, the Rvalue computed on the CU is broadcast to the PE's and stored at the Lvalue location on the PE's. For a SIMD mode assignment of a poly Rvalue to a mono Lvalue, it is undefined to have more than one PE selected. (PE selection is discussed in Sections V and VI.) When such an assignment is defined, the unique selected PE sends the Rvalue it computed to the CU, where it is stored at the Lvalue location.

ELP assignment statements in SPMD mode are performed locally within each of the PE's, except for assigning a poly Rvalue to a mono Lvalue. When such an assignment is performed, a barrier is forced across all PE's so that a unique selected PE can broadcast the Rvalue it computed to all the other PE's; the Rvalue is then stored at the Lvalue location on each of the other PE's. Similar to the SIMD mode case, a SPMD mode assignment of a poly Rvalue to a mono Lvalue is undefined if more than one PE is selected. In SPMD mode, PE selection is done via a conditional statement (Section V) or a selector statement (Section VI) that allows only one PE to write to the mono variable under consideration.

In ELP, it is required that all variables in an Rvalue expression possess the same variable class (i.e., variable classes

TABLE I
Code Generation for ELP Assignment Statement in SIMD Mode

	SIMD mode	Rvalue	
		mono	poly
Lvalue	mono	assignment is performed within the CU	Rvalue from exactly one selected PE is sent to CU and stored at Lvalue location
	poly	mono Rvalue on CU is broadcast to PEs and stored at Lvalue location	assignment is performed within each PE

TABLE II
Code Generation for ELP Assignment Statement in SPMD Mode

	SPMD mode	Rvalue	
		mono	poly
Lvalue	mono	assignment is performed within each PE	force barrier across PEs and b'cast unique selected PE's Rvalue to Lvalue location on other PEs
	poly	assignment is performed within each PE	assignment is performed within each PE

must either all be mono or all be poly). If such a constraint did not exist, variables would have to be implicitly cast from one variable class to the other during the course of an expression evaluation. In SIMD mode this would entail the transfer of data from the CU to the PE's or vice versa, and in SPMD mode a barrier across all PE's is required when converting to mono. Therefore, in both execution modes, the overhead for performing such an implicit cast can be quite expensive. Due to the fact that ELP is an explicit language, casts between mono and poly values are required to be explicitly specified at the assignment level rather than implicitly at the operation level. Forcing explicit casting helps make the programmer more aware of these operations to facilitate their avoidance, when it is possible, when they are costly.

C. Constants

Constants in ELP do not have an explicit variable class associated with them. The ELP compiler implicitly determines (via a semantic analysis of the intermediate code) the variable class of a constant based on its context. Even though ELP is an explicit language, its support of constants does not involve any execution-time overhead and it is very straightforward for the programmer. The latter is shown with the help of examples.

Given the following ELP code segment, the compiler would implicitly determine that the variable class of the constant was mono due to the fact that the variable class of the variable m is mono and because variables in an Rvalue expression must possess the same variable class.

```
mono int m;
poly int p;

p = m + 4;
```

If the above code segment was executed in SIMD mode, the constant would be required on the CU; for SPMD mode, the constant would be required on the PE's.

If the Rvalue expression contains constants and no variables, the compiler assigns the constants the variable class of the Lvalue. Within the next ELP code segment, the string constant is assigned the variable class mono (which is the variable class of the Lvalue); therefore, if the assignment below was executed in SIMD mode the string constant would be allocated only on the CU and if the assignment was executed in SPMD mode the string constant would be allocated only on each PE.

```
mono char *m;

m = "Hello";
```

If an ELP program executes solely in SPMD mode, all constants will be stored on the PE's. If an ELP program executes some code in SIMD mode, constants could be stored on both the CU and on the PE's.

D. Update Statements

For joint SIMD/SPMD operation, a mono variable can have two distinct values—one on the CU and one on the PE's—because all mono variables have storage allocated on the CU for when they are accessed in SIMD mode and storage allocated on the PE's for when they are accessed in SPMD mode. The user is forced to keep track of these "dual" values because ELP is an *explicitly* parallel language. Often times a programmer will want to compute a mono value in a particular mode (SIMD or SPMD) and then want to "update" the "other value" before switching to the "other mode." The static rule for when an update is necessary is as follows. If a mono is used as an Lvalue in SIMD mode and at a subsequent time as an Rvalue in SPMD mode (or vice versa), it must be updated.

The syntax for an *update* statement is the following: "`update` *list-of-mono-variables*;". If this statement is executed in SIMD mode, each of the listed mono variable's CU value is transferred to all the PE's to update the PE (SPMD) value. Similarly, if this statement is executed in SPMD mode, each of the listed mono variable's PE value is transferred to the CU to update the CU (SIMD) value. In the latter case, at least one PE must be selected (discussed in Sections V and VI); if two or more PE's are selected, an arbitrary selected PE's value is transferred (all PE's are guaranteed to have the same value).

Due to the implementation of mono variables in SPMD mode (each PE maintains an identically-valued copy of each mono variable), a concern arises that the scalar-valued property of mono variables in SPMD mode could possibly be corrupted during the course of SPMD execution. By construction, an ELP assignment statement employed in SPMD mode ensures that when mono variables are used as Lvalues, the mono variables always remain scalar-valued (i.e., the mono variable's value on all PE's is identical). Similarly, an `update` statement that changes the PE (SPMD) version of a mono variable, by construction, maintains its scalar-valued integrity. Because the only way to change the value of a mono variable in ELP is via an assignment statement or via an `update` statement, in SPMD mode (and trivially in SIMD mode) all mono variables are guaranteed to remain scalar-valued.

To ensure that mono variables are identically-valued across all PE's in SPMD mode, the use of mono variables as Lvalues in SPMD mode is disallowed within the scope of a poly conditional statement (Section V) or a selector statement (Section VI). For example, the following is not allowed because it will violate the scalar-valued integrity of the mono variable m1:

```
poly int p;
mono int m1, m2, m3;

if (p > 0)
  m1 = m2;
else
  m1 = m3;
```

This constraint does not apply to SIMD mode because mono variables are stored on the CU when executing in SIMD.

E. Array Indexing

Arrays can be declared as either mono or poly and a pointer variable pointing into the array would need to be of the same

TABLE III
CODE GENERATION FOR ELP INDEXING OPERATION IN SIMD MODE

SIMD mode		Offset Expression	
		mono	poly
Base	mono	indexing operation is performed within the CU	send unique selected PE's offset to CU and perform indexing operation within CU
	poly	mono offset expression is broadcast to each PE and indexing operation is performed within each PE	indexing operation is performed within each PE

TABLE IV
CODE GENERATION FOR ELP INDEXING OPERATION IN SPMD MODE

SPMD mode		Offset Expression	
		mono	poly
Base	mono	indexing operation is performed within each PE	force barrier across PEs, b'cast unique selected PE's offset to other PEs, perform indexing within each PE
	poly	indexing operation is performed within each PE	indexing operation is performed within each PE

variable class as the array. A pointer variable used to access an array is considered to be a base address variable. Code generation for an array indexing operation in ELP is dependent on the execution mode, the variable class of the base address variable (either an array variable or a pointer variable as is allowed in the C language), and the variable class of the offset expression. Tables III and IV describe the code generation for the array indexing operation performed in SIMD mode and SPMD mode, respectively. The code generated for ELP indexing operations in both SIMD mode and SPMD mode is analogous to that generated for ELP assignment statements in both modes.

Consider indexing operations in SIMD mode (Table III). If the base is mono, the array elements are stored on the CU, and if the base is poly, the array elements are stored on the PE's. ELP indexing operations in SIMD mode require that the base and the result of the offset expression either both be on the CU or both be on each of the PE's before the indexing operation can take place. When the base and the offset expression are both mono, a standard indexing operation is performed within the CU. Similarly, when the base and the offset expression are both poly, a standard indexing operation instruction is broadcast to (and subsequently performed within) each PE. When the base has variable class poly and the offset expression has variable class mono, first the value of the offset expression computed on the CU is automatically broadcast to the PE's, and then instructions for performing the indexing operation with the CU computed offset are broadcast to the PE's. For an SIMD mode indexing operation consisting of a mono base coupled with a poly offset expression, it is undefined to have more than one PE selected. (PE selection is discussed in Sections V and VI.) When such an indexing operation is defined, the operation is performed within the CU once the selected PE has sent the offset value to the CU.

ELP indexing operations in SPMD mode (Table IV) are, except for one case, performed locally on each of the PE's. Similar to the SIMD case, an SPMD mode indexing operation consisting of a mono base coupled with a poly offset expression is undefined if more than one PE is selected. When such an indexing operation is defined, a barrier is forced across all PE's so that the unique selected PE can broadcast the poly offset it computed to all the other PE's; the indexing operation is then performed within each PE.

F. Inter-Processor Communication

All inter-processor communication in ELP is handled with library routines. Because ELP is uniform, all inter-processor communication library routines must have an SIMD version and an SPMD version that are functionally equivalent. Consequently, inter-processor communication library routines in SIMD mode utilize network permutations, while the SPMD mode versions utilize one-to-one network connections. Further details are outside the scope of this paper.

G. Single-Mode Operation

For programs that execute entirely in SIMD mode, mono variables are allocated with a CU (SIMD) value and without a PE (SPMD) value, and Tables II and IV no longer apply. Similarly, for programs that execute entirely in SPMD mode, mono variables are allocated with a PE (SPMD) value and without a CU (SIMD) value, and Tables I and III are no longer applicable.

Along with ELP programs that are explicitly specified to operate solely in SIMD mode or solely in SPMD mode, the ELP compiler will provide users with an option for default single-mode operation. Given any ELP program, the compiler can be instructed to generate SIMD code solely or SPMD code solely, independent of any `simd` or `spmd` execution mode keywords (see Section VII) used within the program.

V. JOINT SIMD/SPMD DATA-DEPENDENT CONTROL-FLOW CONSTRUCTS

This section discusses control-flow constructs that are data-dependent (i.e., based on the values of data items), whereas Section VI discusses control-flow constructs that are based solely on PE numbers (addresses). The following terms are utilized by both Sections V and VI. PE's that are specified to participate in the execution of a particular code block, independent of execution mode, are *selected PE's*. A *PE selection scope* is a block of code that is to be executed by a certain set of selected PE's, independent of execution mode.

TABLE V
CODE GENERATION FOR ELP IF STATEMENT

if statement	Variable Class of Conditional Expression	
	mono	poly
SIMD mode	if statement is performed within the CU	compute poly conditional – b'cast PEs where true "then" before b'casting PEs where false "else"
SPMD mode	if statement is performed within each PE	if statement is performed within each PE

The two basic control-flow constructs for PE selection in ELP (i.e., *PE selection constructs*) are the if statement employing a poly conditional expression (addressed in this section) and the selector statement (addressed in the next section). Both of these constructs are applied to either one or two static (i.e., known at compile-time) PE selection scopes.

In the C language, data-dependent control-flow constructs include the if, while, do, and for statements. ELP supports control-flow statements such as if's, while's, do's, and for's by applying different parallel interpretations to them depending on the variable class of the conditional expression and the execution mode. Only the if and while statements will be discussed; the do and for statements in ELP are closely related to the ELP while statement (the same is true in the C language). As with Rvalue expressions in ELP, variables within conditional expressions are required to possess the same variable class (i.e., variable classes must either all be mono or all be poly). The reasoning here is the same as that used in Section IV-B concerning the constraint on Rvalue expressions.

A. If Statements

Code generation for the if statement in ELP, as shown in Table V, is dependent on the execution mode and the variable class of the conditional expression. The syntax for an if statement in ELP is the same as for an if statement in the C language.

For a mono conditional expression in SIMD mode, a standard if statement is performed within the CU. Similarly, for a mono or a poly conditional expression in SPMD mode, a standard if statement is performed within each PE.

For the last case in Table V, a poly conditional expression in SIMD mode, an if statement is generated that performs data-dependent masking by executing the "then" clause PE selection scope with all PE's that satisfy the poly conditional (i.e., all initially selected PE's), succeeded by executing the "else" clause PE selection scope with all PE's that do *not* satisfy the poly conditional (i.e., all initially unselected PE's). One implementation for the support of nesting in this case is to maintain an *activity stack* on each PE whose top-of-stack contains the current PE selection bit.[1] If the top-of-stack bit for a PE's activity stack is cleared (= 0) and an instruction is broadcast to the PE, the PE is unselected and does *not* execute the instruction. Similarly, if the top-of-stack bit for a PE's activity stack is set (= 1) and an instruction is broadcast to the PE, the PE is selected and executes the instruction. Thus, the activity stacks are execution-time stacks that are used to keep track of the data-dependent PE selection patterns that occur at different control-flow nesting levels during the course of an ELP program. It is assumed that *all* PE's in the submachine perform the push and pop operations associated with the activity stacks, independent of their selection status.

To demonstrate the management of the activity stacks, consider a poly conditional if statement that possesses an "else" clause PE selection scope. The CU will command the FU to broadcast the following sequence of instructions to the PE's. The "AND" operations below are for handling nested conditionals.

1) perform the poly conditional expression so that a data-dependent PE selection bit is produced within each PE;
2) copy (not remove) the top of the activity stack into "temp," logically AND "temp" with the complement of the (just computed) PE selection bit, and push the result onto the activity stack (this result will control which PE's execute the "else" clause);
3) logically AND "temp" with the PE selection bit and push the result onto the activity stack (this result will control which PE's execute the "then" clause);
4) the "then" clause;
5) pop and discard the top of the activity stack;
6) the "else" clause;
7) pop and discard the top of the activity stack.

This stack management scheme is more fully described in [32].

B. While Statements

Similar to the if statement, code generation for the while statement in ELP, as shown in Table VI, is dependent on the execution mode and the variable class of the conditional expression. For a mono conditional expression in SIMD mode, a standard while statement is performed within the CU. Similarly, for a mono or a poly conditional expression in SPMD mode, a standard while statement is performed within each PE.

For the last case in Table VI, a poly conditional expression in SIMD mode, the size of the set of selected PE's decreases monotonically with the number of loop iterations of the while statement as a result of the conditionals between loop iterations. When the set of selected PE's is null, the while statement terminates. This interpretation of the while statement obviously requires the use of the activity stacks and has been proposed in SIMD languages such as Actus [37]–[39] and Parallel-C [28].

[1]This corresponds to the planned PASM prototype hardware enhancements [32], and differs somewhat from the current prototype implementation of this construct.

TABLE VI
CODE GENERATION FOR ELP WHILE STATEMENT

while statement	Variable Class of Conditional Expression	
	mono	poly
SIMD mode	while statement is performed within the CU	conditional tests are performed in each iteration – number of enabled PEs decreases monotonically
SPMD mode	while statement is performed within each PE	while statement is performed within each PE

For each iteration of a poly conditional `while` statement, the CU will command the FU to broadcast the following sequence of instructions to the PE's (recall that *all* PE's must perform the push and pop operations):

1) perform the current iteration's poly conditional expression so that a data-dependent PE selection bit is produced within each selected PE;
2) pop the current top of the activity stack into "temp," logically AND "temp" with the current iteration's (just computed) PE selection bit, and push the result onto the activity stack;
3) the "body" of the `while` statement.

The CU orchestrates the loop iterations and therefore needs to know when all of the PE's are unselected so that it can terminate the statement. Because the PE selection information is held within the activity stacks on the PE's, special hardware for an "if-any" operation is required to permit the CU to obtain this information.

Code generated for ELP `do` and `for` statements is very similar to that generated for the ELP `while` statement. The syntax for `while`, `do`, and `for` statements in ELP is the same as for `while`, `do`, and `for` statements in the C language.

VI. JOINT SIMD/SPMD PE-ADDRESS DEPENDENT CONTROL-FLOW CONSTRUCTS

Discussion in the previous section centered around control-flow constructs that are data-dependent (i.e., based on the value of data items). This section considers control-flow constructs that are based solely on PE numbers (addresses). Assuming the N PE's in a system are numbered (addressed) from 0 to $N-1$, the PE numbers are used as a basis for selecting PE's.

A. Selector Statements

The construct that ELP uses to support PE-address dependent control-flow is the *selector statement*. The PE selection scope for selector statements is the block of code immediately following the statement, and the statement's syntax and semantics are independent of the execution mode in which it is called. For $N = 2^n$ PE's addressed 0 to $N-1$, the specification format used is an n-position *PE-address mask* [48], [50] consisting of 1's, 0's, and X's (don't care's) that selects all PE's whose physical address matches the PE-address mask. PE-address masks are a convenient notation for specifying PE selection patterns on *large-scale* parallel machines; examples of their use in the specification of common parallel algorithms are provided in [32], [49]–[51]. In SIMD mode, selector statements resemble the selector concept proposed in Parallel-C [28].

The syntax for a selector statement that selects the even-numbered PE's (for $N = 16$) is: "`[X X X 0];`". Within a PE-address mask, a repetition factor can be applied to a particular position. For instance, by using a repetition factor of 3 applied to the `X`, the following selector statement is equivalent to the previous one: "`[{3}X 0];`". The use of mono variables as repetition factors is currently under study. Negative PE-address masks select all PE's whose numbers do not match the mask; they are specified like regular PE-address masks except prepended with a minus sign. Given 16 PE's numbered 0 to 15 executing in SIMD mode, the following code segment can be used to store the value of PE 5's poly integer variable `p` into the CU's mono integer variable `m`:

```
mono int m;
poly int p;

[0 1 0 1];
{
  m = p;
}
```

As with data-dependent PE selection, PE-address dependent selection scopes can be nested (and a PE is selected only if it is selected by all PE-address masks in the nesting).

Code generation for the selector statement is based on the execution mode in effect when the selector statement is called. In SPMD mode, each PE (independently) determines at execution-time whether its address matches the PE-address mask used in the selector statement. If a PE determines that it has a match (i.e., the PE is selected), the PE proceeds to execute the corresponding PE selection scope (i.e., block of one or more ELP statements); if a PE determines that it does not have a match (i.e., the PE is not selected), the PE proceeds to branch around the corresponding PE selection scope.

One way to implement this is as follows. The compiler generates an n-bit value M, where bit $i = 0$ if position i of the mask is X, and bit $i = 1$ otherwise. The compiler also generates an n-bit value Q, where bit i is the value of position i of the mask for each non-X position, and bit $i = 0$ otherwise. It is assumed that each PE knows its own number, and at execution-time, each PE uses its own PE number, NUM, to compute (either in hardware or software) $S = (NUM \oplus Q) \wedge M$. If the result is zero, the PE has a match and is selected.

Consider the following code segment for $N = 16$:

```
[X X 0 1];
{
  /* PE selection scope */
}
```

Specifically, focus on the operation of PE $9 = (1001)_2$ as it executes the code segment. First, the compiler determines via the PE-address mask `[X X 0 1]` that $M = 0011$ and that $Q = 0001$. Then at execution-time with $NUM = 1001$, PE 9 will generate S as follows: $S = (1001 \oplus 0001) \wedge 0011 = 0000$. Thus, PE 9 does have a match and is selected to execute the PE selection scope.

In SIMD mode, when both data-dependent selection (discussed in the previous section) and PE-address dependent selection are used, a PE is active only if it is selected by both selection schemes. For example, if $N = 8$, and only in PE's 0 and 1 is the poly variable $A > 0$, then given the code:

```
[X X 0];
if (A > 0)
{
  /* PE selection scope */
}
```

only PE 0 is selected by both schemes and executes the PE selection scope. One way to implement this is to broadcast the M and Q representation of the PE-address mask to the PE's and all currently selected PE's compute (in hardware or software) $S = (NUM \oplus Q) \wedge M$. Then the above example is processed by effectively executing:

```
if (S == 0) {
  if (A > 0) {
    /* PE selection scope */
  }
}
```

performing the activity stack manipulations described in the previous section.

Due to the small size of the PASM prototype, the CU converts PE-address masks into 16-bit vectors, where bit i is 1 if PE i is to be selected, and PE i is sent bit i of the vector. For large N, this approach is inappropriate.

As stated in Section IV, selector statements are dependent on the extent of parallelism that the user intends to employ during program execution. In general, given that an ELP program will be executed on a machine or independent submachine with $P = 2^p$ PE's, all selector statements used in the program must contain exactly p PE-address mask positions (1's, 0's, or X's). If the value of p changes, the selector statements must be modified accordingly so that the program's parallel execution scales properly.

B. Multiselect Statements

A multiway selector statement incorporates the use of multiple PE-address masks into a single language construct. The syntax of the statement is demonstrated with the following example (assuming $N = 16$):

```
poly int p;

multiselect {
  [X X X 0] : p = 0;
  [X X 0 1] : p = 1;
              break;
  default   : p = 2;
}
```

As can be seen, the `multiselect` statement is very similar to the `switch` statement in the C language. However, where the `switch` statement has integer cases, the `multiselect` statement uses PE-address masks. Like the `switch` statement, the `multiselect` statement permits control to proceed through the cases until a `break` statement is reached.

Each PE tries to match its address with the given PE-address masks in the order specified within the `multiselect` statement. Once a PE's address matches a PE-address mask, the PE executes the associated statements (i.e., the associated PE selection scopes) until it reaches a `break` statement or until it reaches the end of the `multiselect`. Thus, within a `multiselect` statement, a PE executes the ELP statements associated with the PE-address mask it matched and those below that until encountering either a `break` statement or the end of the `multiselect`. When it encounters a `break` statement, it exits the `multiselect` statement. In the above example, all even-numbered PE's will execute the "`p = 0;`" statement. The set of PE's that will execute the "`p = 1;`" statement are all PE's that either match PE-address mask `[X X X 0]` or match PE-address mask `[X X 0 1]`. The `default` statement is executed by all PE's that have not executed a `break` statement. In the above example, the set of PE's that will take the `default` are all PE's that match the PE-address mask `[X X 1 1]`.

Code generation for the `multiselect` statement, like the selector statement, depends on the execution mode in effect when the statement is called. In SPMD mode, code generation amounts to performing a series of selector statements (one for each PE-address mask). Each PE needs to maintain a flag (initialized to false) that indicates whether its number has matched a PE-address mask (flag is true) or not (flag is false). If the flag is false, a PE branches over that mask's associated statements and to the next PE-address mask. If the flag is true, a PE executes the `multiselect`'s statements (beginning with those associated with the mask it matched) until it reaches a `break` statement (or until it reaches the end of the `multiselect`). Once a `break` statement is reached, a PE branches to the end of the `multiselect.` If a PE reaches the `default` (because it has not executed a `break` statement), it proceeds to execute the statements associated with the `default.`

The only difference between the SIMD mode implementation and the SPMD mode implementation is that instead of "branching over" instructions or statements that are not to be executed, the PE's ignore them even though they are broadcast. The SIMD implementation of the `multiselect` statement is presented algorithmically. Each PE needs to maintain two single bit flags: one ("sflag") that indicates whether it has matched a PE-address mask (flag is one) or not (flag is zero), and one ("bflag") that indicates whether it has been broadcast a `break` statement (flag is zero) or not (flag is one). Let "top" be the single bit value currently on top of a PE's local activity stack and let "push x" and "pop" be the local activity

```
[ sflag ← 0 ]
[ bflag ← 1 ]
for (every PE-address mask $i$ from top to bottom) {
    [ computation of $S_i$ ]
    [ sflag ← sflag ∨ $\bar{S}_i$ ]
    [ push (top ∧ sflag ∧ bflag) ]
    [ $scope_i$ ]
    if (break encountered at end of $scope_i$)
       [ bflag ← 0 ]
    [ pop ]
}
```

Fig. 1. Code generation for `multiselect` statement in SIMD mode.

stack operations of push the single bit value "x" and pop, respectively. From Section V-A, it is assumed that *all* PE's perform the local activity stack "push x" and "pop" operations, even if currently unselected. The `default` is treated as the bottom mask `[{n}X]` (for $N = 2^n$ PE's) within the `multiselect`. Lastly, let S_i and $scope_i$ be the "S" value (defined in Section VI-A) and PE selection scope, respectively, for the ith PE-address mask in the `multiselect` from top to bottom. Without loss of generality, assume that any `break` statement in $scope_i$ must occur as the last statement.

Fig. 1 depicts the SIMD implementation algorithm for the `multiselect` statement. If a statement is enclosed in square brackets, it signifies that the CU is to broadcast the statement's instructions to all selected PE's. Because a `multiselect` statement can be nested within another PE selection construct, there may be unselected PE's prior to starting the `multiselect`'s execution. First, the CU broadcasts (to all selected PE's) instructions for initializing "sflag" to zero and "bflag" to one. Then, the CU iterates over all consecutive PE-address masks from top to bottom within the `multiselect`. (Recall that the bottom PE-address mask is the `default`'s.) For PE-address mask i, the following is performed. The CU broadcasts (to all selected PE's) instructions for computing S_i and for storing in "sflag" the logical OR of "sflag" and $\bar{S}_i$. Therefore, a PE's "sflag" value is one only if the PE has matched at least one of the PE-address masks from the top to i. (Recall from Section VI-A that a PE is selected only if "S" equals zero.) Broadcasting (to *all* PE's) a push of the logical AND of both flags and the top-of-stack value selects only those PE's who: 1) were selected prior to entering the `multiselect`, 2) have matched at least one PE-address mask, and 3) have not yet encountered a `break` statement. Such a selection pattern is the one required when broadcasting $scope_i$'s instructions to the PE's. If, when broadcasting $scope_i$'s instructions, a `break` statement is encountered by the CU, the CU immediately broadcasts (to all selected PE's) instructions for setting "bflag" to zero. Based on the previously mentioned push operation, if a PE's "bflag" value is zero, the PE will be unselected during the broadcasts of all remaining $scope_i$'s. The ith iteration is concluded after a pop of the current PE selection status is broadcast (to *all* PE's). This restores the PE selection status in effect prior to entering the `multiselect`.

VII. SIMD/SPMD Execution Mode Specification

This section describes execution mode specification for a program that uses both SIMD mode and SPMD mode and switches between the two modes one or more times at instruction level granularity. More information on SIMD/SPMD execution mode management in ELP can be found in [34].

SIMD/SPMD execution mode specification is statically scoped and uses the keywords `simd` and `spmd`. Specification can be done on a per-block basis as

```
simd {...}          spmd {...}
```

or on a per-function basis when a function is declared as

function-name`((...)): simd` *function-name*`((...)): spmd`
`{...}` `{...}`

Execution mode specifiers can be nested and are kept track of with a compile-time stack. Recall that all PE's in a machine (or in an independent submachine if the system is partitionable) must be in the same mode.

To support `simd` functions and `spmd` functions in ELP, execution-time stacks for parameters and local variables must be maintained both on the CU and on each PE. Specifically, poly parameters and poly locals are pushed onto each PE's execution-time stack (independent of execution mode); mono parameters and mono locals within an `simd` function are pushed onto the CU's execution-time stack; and mono parameters and mono locals within an `spmd` function are pushed onto each PE's execution-time stack. Both `simd` and `spmd` functions can be executed within the scope of control-flow constructs that select a subset of PE's to execute a block of ELP code.

VIII. Overview of the Compilation/Linking/Loading Process

Fig. 2 provides an outline of the process for going from an ELP source file to downloading machine code onto the PASM prototype. Boxes are utility programs and ellipses are intermediate code files. Solid arrows indicate utility program input and output files. A dashed arrow indicates that only the file's symbol table is input to the utility program.

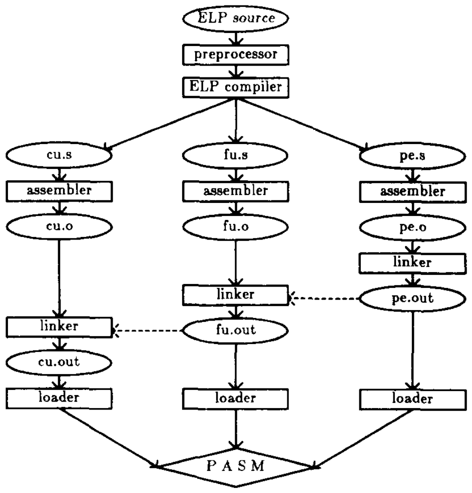


Fig. 2. Flow chart going from ELP source to PASM.

```
#define IN_SIZE 100            /* input signal size */

poly int pInSignal[ INSIZE ];   /* input signal      */
poly int pOutSignal[ INSIZE-2 ];/* output signal     */

MedianFilter(): simd
{
mono int mPos;           /* current window position */
poly int pW1, pW2, pW3; /* current three window points */
  for (mPos = 1; mPos < (IN_SIZE-1); mPos = mPos + 1) {
    pW1 = pInSignal[ mPos - 1 ]; /* obtain */
    pW2 = pInSignal[ mPos ];     /* window */
    pW3 = pInSignal[ mPos + 1 ]; /* values */
    update mPos;
    spmd { /* determine median window value */
      if ( ((pW1 <= pW2) && (pW2 <= pW3)) ||
           ((pW3 <= pW2) && (pW2 <= pW1)) )
        pOutSignal[ mPos ] = pW2;
      else if ( ((pW2 <= pW1) && (pW1 <= pW3)) ||
                ((pW3 <= pW1) && (pW1 <= pW2)) )
        pOutSignal[ mPos ] = pW1;
      else
        pOutSignal[ mPos ] = pW3;
    }
  }
}

main(): simd
{
  MedianFilter();
}
```

Fig. 3. Sample ELP program for performing median filtering.

Before being input to the ELP compiler, an ELP source file is first modified by a standard C language preprocessor, which substitutes in constant values and expands macros. The ELP compiler generates three distinct serial assembly source files, each of which is assembled using the standard Unix System V 68000 assembler. One file is for the CU and contains SIMD mono global variable storage and SIMD mono variable computation code, another file is for the FU and contains SIMD poly variable computation code, and the last file is for the PE's and contains SPMD mono global variable storage, SIMD/SPMD poly global variable storage, and all SPMD computation code. Data files to be processed are loaded into the PE's by the memory management system.

After the three distinct assembly source files have been assembled, the resulting three object files need to go through the linking process totally ordered in time. Specifically, the PEs' object file must be linked before the FU's object file because the SIMD poly variable computations need to know the addresses of the SIMD poly variables. The FU's object file must be linked before the CU's object file because the CU needs to know the addresses of the SIMD poly code blocks it will be sending from the FU memory to the FU queue. Thus, the symbol table information pertaining to the PE memory image is required during the FU's linking process, and the resulting symbol table information pertaining to the FU memory image is required during the CU's linking process. (The standard Unix System V 68000 linker has an option for incorporating an object file's symbol table while ignoring its code and data segments.) After the three distinct object files have each been linked, the resulting CU, FU, and PE memory image files are downloaded onto the CU, FU, and all PE's, respectively, within the PASM prototype, and execution can commence.

All library routines have an SIMD version (invoked by the CU) and an SPMD version (invoked by the PE's) that are semantically equivalent. Therefore, an ELP library is decomposed into three portions: a CU portion consisting of the SIMD routines' mono code/data, an FU portion consisting of the SIMD routines' poly code, and a PE portion consisting of the SIMD routines' poly data and the SPMD routines' mono code/data and poly code/data. Each portion of the library is linked during the corresponding linking process in Fig. 2.

The ELP compiler's scanner and parser were generated using the Purdue Compiler Construction Tool Set [36]. The ANTLR parser-generator included within the tool set builds a top-down parser and provides a flexible interface for manipulating inherited attributes, as well as a clean, concise, notation for action and rule specification.

IX. Sample ELP Program—Median Filtering

Median filtering is a one-dimensional windowing operation where for each position the window is applied across some digital input signal: the corresponding digital output signal value is simply the median value within the current input window. Fig. 3 illustrates a parallel median filtering program where an input signal is subdivided among all of the PE's in the submachine. Due to the fact that the program does not contain any PE-address dependent control-flow constructs (i.e., selector statements or `multiselect` statements), the desired submachine size does not need to be specified until load-time. Each PE computes its local portion of the output signal by applying a window of size three to its local portion of the input signal. The PE's function independently from one another because it is assumed that their local portions of the input signal are overlapped on both ends by two input signal points (as a result of a prior inter-PE data transfer).

To keep track of whether a variable is mono or poly, the following variable naming convention is employed (although not required): all mono variable names are prepended with a lower case "m" and all poly variable names are prepended with a lower case "p." Both the signal arrays `pInSignal` and `pOutSignal` along with the current three window values `pW1`, `pW2`, and `pW3` are declared to be poly variables because they all will be instantiated with different values on different PE's. The windowing index `mPos` is declared a mono variable due to the fact that the windowing will proceed in an identical fashion across all PE's, independent of execution mode.

The `MedianFilter` routine starts and finishes in SIMD mode due to the declaration: "`MedianFilter(): simd`." Both the `for` statement employing a mono conditional expression and the indexing necessary for obtaining the current three window values are performed in SIMD mode. Because in SIMD mode `pInSignal` is stored on the PE's and `mPos` is stored on the CU, the array indexing operation "`pInSignal[mPos - 1]`" must broadcast the value of the mono expression "`mPos - 1`" to the PE's so that they can use the value to offset into `pInSignal` (this was introduced in Section IV-E). Within the routine, only the data conditionals used to determine the median value within each window are performed in SPMD mode. An `update` statement is used to transfer the current value of `mPos` from the CU to the PE's in preparation for switching from SIMD mode to SPMD mode via the `spmd { . . . }` encapsulation.

X. Conclusion

When trying to harness parallelism, it is useful to be able to employ multiple modes of parallelism within an application [19]. Features of an explicitly parallel language and corresponding compiler that combine the parallelism modes of SIMD and SPMD were presented. The proposed language is completely uniform with respect to the SIMD and SPMD modes of parallelism, is capable of switching modes at instruction level granularity, and is to be supported by a full native-code compiler (currently under development). Due to the language's uniform nature with respect to SIMD mode and SPMD mode, a given parallel program can be compiled to execute solely in SIMD mode (ignoring all mode specifiers), solely in SPMD mode (ignoring all mode specifiers), or utilizing both modes (adhering to all mode specifiers). Such a parallel language is important because: 1) it provides users with the ability to explicitly control different facets of mixed-mode parallel processing systems; 2) it serves as a foundation for the design of a compiler that automatically determines and specifies the best mode for code segments; 3) it simplifies mixed-mode programming by employing a single program model for both modes of parallelism; and 4) it facilitates the use of reconfiguration for fault tolerance.

Specific language features addressed included data management, data-dependent control-flow, and PE-address dependent control-flow. For each of these language features, a clear specification and an efficient implementation were provided. These features were developed as a result of experiences programming the prototype. All language concepts presented herein can be applied to distributed-memory machines capable of SIMD, SPMD, or SIMD/SPMD operation.

There is a great deal that needs to be learned about the programming and design of mixed-mode parallel computers. An explicitly parallel language, such as ELP, provides a vehicle for the exploration of and experimentation with mixed-mode parallelism, and thus aids in this learning process.

Acknowledgment

The authors acknowledge many useful discussions with W. Cohen, S.-D. Kim, C. Krauskopf, J. Kuehn, W. Nation, T. Parr, P. Pero, D. Quammen, and J. Stuhlmacher.

References

[1] Ames Research Center, *CFD: a FORTRAN-based language for Illiac IV*, NASA, 1974.
[2] M. Auguin and F. Boeri, "The OPSILA computer," in *Parallel Languages and Architectures*, M. Consard, Ed. Holland, Elsevier Science, 1986, pp. 143–153.
[3] M. Auguin, F. Boeri, J. P. Dalban, and A. Vincent-Carrefour, "Experience using a SIMD/SPMD multiprocessor architecture," *Microprocessing and Microprogramming*, vol. 21, Aug. 1987, pp. 171–177.
[4] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, no. 8, pp. 746–757, Aug. 1968.
[5] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 836–844, Sept. 1980.
[6] ——, "Bit serial parallel processing systems," *IEEE Trans. Comput.*, vol. C-31, no. 5, pp. 377–384, May 1982.
[7] T. B. Berg and H. J. Siegel, "Instruction execution trade-offs for SIMD vs. MIMD vs. mixed-mode parallelism," in *Proc. Int. Parallel Processing Symp.*, May 1991, pp. 301–308.
[8] T. Blank, "The MasPar MP-1 architecture," in *Proc. IEEE Compcon*, Feb. 1990, pp. 20–24.
[9] E. C. Bronson, T. L. Casavant, and L. H. Jamieson, "Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system," *IEEE Trans. Parallel Distributed Syst.*, vol. 1, no. 2, pp. 195–205, Apr. 1990.
[10] J. C. Browne, A. R. Tripathi, S. Fedak, A. Adiga, and R. N. Kapur, "A language for specification and programming of reconfigurable parallel computation structures," in *Proc. 1982 Int. Conf. Parallel Processing*, Aug. 1982, pp. 142–149.
[11] P. Christy, "Software to support massively parallel computing on the MasPar MP-1," in *Proc. IEEE Compcon*, Feb. 1990, pp. 29–33.
[12] C. L. Cline and H. J. Siegel, "Augmenting Ada for SIMD parallel processing," *IEEE Trans. Software Eng.*, vol. SE-11, no. 9, pp. 970–977, Sept. 1985.
[13] W. Crowther, J. Goodhue, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on a 128-node butterfly parallel processor," in *Proc. 1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 531–540.
[14] F. Darema-Rodgers, D. A. George, V. A. Norton, and G. F. Pfister, *Environment and System Interface for VM/EPEX*, Res. Rep. RC11381 (#51260), IBM T. J. Watson Research Center, 1985.
[15] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," *Parallel Comput.*, vol. 7, no. 1, pp. 11–24, Apr. 1988.
[16] P. Duclos, F. Boeri, M. Auguin, and G. Giraudon, "Image processing on a SIMD/SPMD architecture: OPSILA," in *Proc. Ninth Int. Conf. Pattern Recognition*, Nov. 1988, pp. 430–433.
[17] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, "Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting," *J. Parallel Distributed Comput.*, vol. 11, no. 3, pp. 239–251, Mar. 1991.
[18] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901–1909, Dec. 1966.
[19] R. F. Freund, "Optimal selection theory for superconcurrency," in *Proc. Supercomput. '89*, Nov. 1989, pp. 699–703.
[20] J. P. Hayes and T. N. Mudge, "Hypercube supercomputers," *Proc. IEEE*, vol. 77, no. 12, pp. 1829–1841, Dec. 1989.
[21] W. D. Hillis, *The Connection Machine*. Cambridge, MA: M.I.T. Press, 1985.

[22] D. J. Hunt, "AMT DAP-A processor array in a workstation environment," *Comput. Syst. Sci. and Eng.*, vol. 4, no. 2, pp. 107–114, Apr. 1989.

[23] L. H. Jamieson, "Characterizing parallel algorithms," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. Cambridge, MA: M.I.T. Press, 1987, pp. 65–100.

[24] H. F. Jordan, "Structuring parallel algorithms in an MIMD, shared memory environment," *Parallel Comput.*, vol. 3, no. 2, pp. 93–110, May 1986.

[25] ——, "The Force," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. Cambridge, MA: M.I.T. Press, 1987, pp. 395–436.

[26] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[27] S. D. Kim, M. A. Nichols, and H. J. Siegel, "Modeling overlapped operation between the control unit and processing elements in an SIMD machine," *J. Parallel and Distributed Comput.*, Special Issue on Modeling of Parallel Computers, vol. 12, no. 4, pp. 329–342, Aug. 1991.

[28] J. T. Kuehn and H. J. Siegel, "Extensions to the C programming language for SIMD/MIMD parallelism," in *Proc. 1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 232–235.

[29] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randall, "Glypnir-A programming language for Illiac IV," *Commun. ACM*, vol. 18, no. 3, pp. 157–164, Mar. 1975.

[30] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons*. New York: Wiley, 1987.

[31] P. Mehrotra and J. Van Rosendale, "The BLAZE language: A parallel language for scientific programming," *Parallel Comput.*, vol. 5, no. 3, pp. 339–361, Nov. 1987.

[32] W. G. Nation, S. A. Fineberg, M. D. Allemang, T. Schwederski, T. L. Casavant, and H. J. Siegel, "Efficient masking techniques for large-scale SIMD architectures," in *Proc. Frontiers '90: Third Symp. Frontiers Massively Parallel Computation*, Oct. 1990, pp. 259–264.

[33] M. A. Nichols, H. J. Siegel, H. G. Dietz, R. W. Quong, and W. G. Nation, "Eliminating memory fragmentation within partitionable SIMD/SPMD machines," *IEEE Trans. Parallel Distributed Systems*, Special Issue on Parallel Languages and Compilers, vol. 2, no. 3, pp. 290–303, July 1991.

[34] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Execution mode management in an SIMD/SPMD parallel language/compiler," in *Proc. COMPSAC '91: Fifteenth Annu. Int. Comput. Software and Appl. Conf.*, Sept. 1991, pp. 392–397.

[35] S. F. Nugent, "The iPSC/2 direct-connect communications technology," in *Proc. Third Conf. Hypercube Comput. and Appl.*, Jan. 1988, pp. 51–60.

[36] T. J. Parr, H. G. Dietz, and W. E. Cohen, "Purdue Compiler-Construction Tool Set," Tech. Rep. TR-EE 90-14, School of Elec. Eng., Purdue Univ., 1990.

[37] R. H. Perrott, R. W. Lyttle, and P. S. Dhillon, "The design and implementation of a Pascal-based language for array processor architectures," *J. Parallel Distributed Comput.*, vol. 4, no. 3, pp. 266–287, June 1987.

[38] R. H. Perrott, "A language for array and vector processors," *ACM Trans. Programming Languages and Syst.*, vol. 1, no. 2, pp. 177–195, Oct. 1979.

[39] ——, *Parallel Programming*. Reading, MA: Addison-Wesley, 1987.

[40] M. J. Phillip and H. G. Dietz, "Toward semantic self-consistency in explicitly parallel languages," in *Proc. Fourth Int. Conf. Supercomput.*, May 1989, pp. 398–407.

[41] M. J. Phillip, "Unification of synchronous and asynchronous models for parallel programming languages," M.S.E.E. thesis, School of Elec. Eng., Purdue Univ., 1989.

[42] A. P. Reeves, "Parallel Pascal: An extended Pascal for parallel computers," *J. Parallel Distributed Comput.*, vol. 1, no. 1, pp. 64–80, Aug. 1984.

[43] M. D. Rice, S. B. Seidman, and P. Y. Wang, "A high-level language for SIMD computation," in *Proc. CONPAR 88*, Sept. 1988, pp. 384–391.

[44] J. R. Rose and G. L., Jr. Steele, "C*: An extended C language for data parallel programming," in *Proc. Second Int. Conf. Supercomput.*, vol. 2, May 1987, pp. 2–16.

[45] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "Design and implementation of the PASM prototype control hierarchy," in *Proc. Second Int. Conf. Supercomput.*, vol. I, May 1987, pp. 418–427.

[46] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," in *Proc. AFIPS 1980 Nat. Comput. Conf.*, June 1980, pp. 631–641.

[47] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel processing systems," *IEEE Comput. Mag.*, Special Issue on Parallel Processing for Computer Vision and Image Understanding, vol. 25, no. 2, pp. 54–63, Feb. 1992.

[48] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Comput.*, vol. C-26, no. 2, pp. 153–161, Feb. 1977.

[49] ——, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. Comput.*, vol. C-28, no. 12, pp. 907–917, Dec. 1979.

[50] ——, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, second ed. New York: McGraw-Hill, 1990.

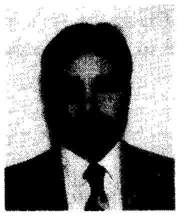
[51] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, no. 12, pp. 934–947, Dec. 1981.

[52] H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture*, D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, Eds. Washington, DC: IEEE Computer Society Press, 1987, pp. 387–407.

[53] R. H. Thomas and W. Crowther, *The Uniform System: An approach to runtime support for large scale shared memory parallel processors*, BBN Advanced Computers, Inc., Cambridge, MA, 1987.

[54] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *IEEE Comput. Mag.*, vol. 21, no. 8, pp. 26–38, Aug. 1988.

[55] P. Y. Wang, S. B. Seidman, M. D. Rice, and T. E. Gerasch, "An object-method programming language for data parallel computation," in *Proc. 22nd Hawaii Int. Conf. Syst. Sci.*, vol. 2, Jan. 1989, pp. 745–750.



Mark A. Nichols (S'86–M'91) received the B.S. degree in 1985 with a triple major of electrical engineering, computer engineering, and mathematics (computer science) from Carnegie Mellon University. In 1986 he received the M.S.E.E. degree from the Georgia Institute of Technology and in 1991 completed the Ph.D. degree at Purdue University in electrical engineering.

He is currently with NCR, San Diego, CA. While at Purdue, he was employed as the project leader for the design and implementation of a parallel language and compiler for the reconfigurable PASM parallel processing system prototype. His research interests include parallel language/compiler design, parallel architecture modeling, and interconnection networks.

Dr. Nichols is a member of the IEEE Computer Society and the Association for Computing Machinery.

Howard Jay Siegel (M'77–SM'82–F'90), for a photograph and biography, see the January issue of this TRANSACTIONS, p. 2.

Henry G. Dietz (M'91) received the B.S., M.S., and Ph.D. degrees in computer science from the Polytechnic Institute of New York.

In 1986, he joined the faculty at Purdue University, West Lafayette, IN, where he is an Assistant Professor of Electrical Engineering (Computer Engineering). He has co-authored over 40 technical papers primarily in compiler optimization/parallelization and parallel architecture. He founded and leads CARP, the Compiler-oriented Architecture Research group at Purdue. Other current research activities include development of PCCTS, the Purdue Compiler-Construction Tool Set, a collection of public domain software tools for construction of optimizing/parallelizing compilers.