JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 21, 271–288 (1994)
Vol. 21, No. 3, June 1994
Special Issue on Heterogeneous Processing

A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments[1]

DANIEL W. WATSON,*[,2] HOWARD JAY SIEGEL,†[,3] JOHN K. ANTONIO,†[,4] MARK A. NICHOLS,‡[,5] AND MIKHAIL J. ATALLAH§[,6]

*Department of Computer Science, Utah State University, Logan, Utah 84322-4205; †Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907-1285; ‡NCR Corporation, San Diego, California 92127-1806; and §Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907-1398

One of the challenges for parallel compilers and compiler-related tools is, given a machine-independent parallel language, to generate executable code for a variety of computational models, and to identify those specific parallel modes for which a program is well-suited. One portion of this problem, developing a method for estimating the relative execution time of a data-parallel algorithm in an environment capable of the SIMD and SPMD (MIMD) modes of parallelism, is presented. Given a data-parallel program in a language whose syntax is mode-independent and empirical information about instruction execution time characteristics, the goal is to use static source-code analysis to determine an implementation that results in an optimal execution time for a mixed-mode machine capable of SIMD and SPMD parallelism. Statistical information about individual operation execution times and paths of execution through a parallel program is assumed. A secondary goal of this study is to indicate language, algorithm, and machine characteristics that must be researched to learn how to provide the information needed to obtain an optimal assignment of parallel modes to program segments. © 1994 Academic Press, Inc.

1. INTRODUCTION

In general, writing effective programs for parallel computers remains a complex and difficult endeavor. In many cases, algorithms that work well under a serial execution model require reformulation for implementation on a parallel system. The problem is compounded by the rich diversity of parallel architectures available, each with its own attributes. Subsequently, parallel languages for these systems vary substantially, as vendors are (justifiably) concerned with maximizing performance for their products. One of the challenges for compilers and compiler-related tools is, given a machine-independent parallel language, to generate executable code for a variety of parallel computational modes, and to identify those specific modes for which the program is well-suited. This problem is even more difficult on a *heterogeneous* system [19, 20] where different parallel modes and/or machines can be used to perform the segments of a single task.

One type of a heterogeneous system is a *mixed-mode* machine, in which the processors are capable of operating in either the SIMD or MIMD modes of parallelism and can dynamically switch between modes at instruction-level granularity with generally negligible overhead, e.g., PASM [36], Triton [33], and OPSILA [3]. The focus here is the use of mixed-mode machines to perform *data-parallel* algorithms, where the data is distributed among the processor/memory pairs of a parallel machine and all processors use the same code to operate on their local data [24]. In MIMD mode, data-parallel algorithms are associated with *SPMD* (single program–multiple data) operation, where all processors are executing the same program, but are doing it asynchronously with respect to one another, i.e., each processor follows its own control path through its copy of the program [12]. The relative execution times of the segments of an algorithm using the SIMD or SPMD modes of computation must be estimated to map a data-parallel algorithm onto a mixed-mode machine in an effective way. Given a data-parallel program written in a language whose syntax is mode-independent and empirical information about instruction characteristics and execution paths, the goal addressed here is to make a static source-code-based determination of the implementation that results in the optimal execution time on a mixed-mode machine.

One aspect of current research in parallel optimization and analysis techniques is concerned with extending traditional compiler optimization techniques, including loop concurrentization, code hoisting, and common subexpression elimination, for parallel programs. Many conventional optimization techniques employed by serial compilers are applicable in parallel compilers; however,

[1] This research was supported by the Office of Naval Research under grant number N00014-90-J-1937, by Rome Laboratory under contract numbers F30602-92-C-0150 and F30602-94-C-0022, by NRaD under contract number N68786-91-D-1799, and by the National Science Foundation under grant number CCR-9202807.

[2] E-mail: watson@cs.usu.edu.
[3] E-mail: hj@ecn.purdue.edu.
[4] E-mail: jantonio@ecn.purdue.edu.
[5] E-mail: mark.nichols@sandiegoca.ncr.com.
[6] E-mail: mja@cs.purdue.edu.

0743-7315/94 $6.00

in some cases, traditional optimization approaches without proper analysis can produce erroneous results [29]. There are a number of research efforts directed specifically toward improving execution time in a parallel environment by performing compile-time analysis of programs. In [22], communication costs in a parallel program are analyzed as a function of array dimension and the number of available processors. A time-cost approach based on analysis and simulation is developed in [34] to determine time-cost behavior of parallel computations based on parameters such as input, algorithm, data structure, execution overhead, and execution environment. In [16], a timing analysis is employed to allow compilers for barrier MIMD machines to eliminate a significant percentage of run-time synchronization. The goal of *code-type profiling* is to identify partitions of a program with similar computational requirements [18]. An "optimal selection framework" is presented in [39] as a model for determining the optimal configuration of a heterogeneous suite of supercomputers to perform each task in a given set of applications.

One area where static source-code analysis may prove beneficial is in the execution of parallel algorithms in an environment capable of both SIMD and MIMD modes of parallelism. Heterogeneous systems exploit the diverse computational requirements of traditional supercomputing problems by the selection and use of different types of machines for the computation required [19, 20, 27]. Typical examples of implementation studies using heterogeneous suites of machines are summarized in [37]. Several studies (e.g., [4, 5, 17, 21]) have examined the implementation of parallel programs in a *mixed-mode* machine, i.e., a machine that can operate in either the SIMD or MIMD mode of parallelism and can dynamically switch between modes at instruction-level granularity with generally negligible overhead. A common result from these studies is that cases exist where the execution time of a parallel application can be reduced by exploiting the mode of parallelism that best matches each portion of the program.

Recent academic and commercial interest in parallel computing systems has increased activity in the development of unifying parallel programming models. New programming languages and refined models of programming for parallel machines form an area of research that benefits from this increased interest. Some examples are "Refined C," CODE, and SUPERB. The "Refined C" parallel language is developed in [15] to provide programmers with a means of expressing algorithms for different parallel computers without imposing a specific parallel programming style. A computation-oriented display environment (CODE) is introduced in [7] that encompasses most existing or proposed MIMD architectures and the programming systems related to them. The SUprenum ParallelizER Bonn (SUPERB) [41] is an interactive system for the semi-automatic transformation of FORTRAN 77 programs into MIMD and SIMD programs by using a catalog of parallelization transformations.

Another approach in the development of unifying programming models is to provide languages that support multiple modes of parallelism. Examples of languages designed to support either the SIMD or SPMD modes of data-parallel programming include CM-Fortran [11], HPF [23], and Fortran-D [25]. Other languages, such as Computation Structures Language (*CSL*) [8], Hellena [4], and Explicit Language for Parallelism (*ELP*) [31], have been developed for machines that are capable of mixed-mode parallelism, and include language elements for both SIMD and SPMD (or MIMD) computation. These "mixed-mode" languages provide support for executing different portions of the same program in different modes (e.g., SIMD versus SPMD). CSL is a PASCAL-like explicitly parallel job control language used on TRAC that supports the high-level specification and scheduling of SIMD and MIMD tasks. Hellena is an explicitly parallel preprocessed version of PASCAL for OPSILA that supports dynamic mode switching at the instruction level. ELP is an example of a language whose goal is for all of the statements and constructs to have functionally equivalent SIMD and SPMD interpretations, allowing segments of the same program to be compiled for different parallel models.

The development of mode-independent languages makes possible the incorporation of the decision-making process for selecting the appropriate parallel mode (heretofore performed by knowledgeable programmers) into the realm of the parallel code compiler. Toward that goal, this study proposes a framework for the static source-code-based analysis of execution time for data-parallel algorithms on a mixed-mode machine. These algorithms are assumed to be written in a mode-independent language, and to be implemented in an SIMD/SPMD environment. By establishing this framework, the three elements described above, compile-time analysis, use of different parallel modes, and unifying programming models, are brought together to provide a basis for writing efficient parallel algorithms. The technique involves transforming the program into an SIMD/SPMD trade-off tree, whose structure represents the scope levels within the program. Information at the leaf nodes of the tree, representing blocks of code in the program, is then combined using rules to arrive at decisions for the best parallel mode in which to implement each portion of the program.

To illustrate underlying concepts for the methods presented, the techniques are first developed for the case where a parallel program is to be implemented in either pure SIMD mode or pure SPMD mode (distributed memory machines are assumed). The more general case of a single program that employs both modes during the course of execution on a mixed-mode machine is then considered. For ease of presentation, the programming

model presented here is restricted to basic processor operations and elementary control-flow constructs. Statistical information about individual operation execution times and paths of execution through a parallel program is assumed. This basic model can be enhanced to include other language constructs.

The techniques presented here are directly applicable to the analysis of programs for implementation in a mixed-mode system. Furthermore, they provide a basis for studying the more general problem of optimizing resources in a multiple-machine heterogeneous computing environment. A secondary goal of this study is to indicate language, algorithm, and machine characteristics that must be researched to learn how to provide the information needed to obtain an optimal assignment of parallel modes to program segments.

The system model and representative mode-independent language assumed in later sections are described in Section 2. Many of the reasons why different parallel implementations exhibit disparate execution time performances are the result of inherent differences in parallel modes. These differences are examined in Section 3. In Section 4, techniques are presented for choosing the single-mode implementation of a parallel program that minimizes execution time. Basic definitions used throughout the rest of the paper are included in Subsection 4.2, and the single-mode framework itself is developed in Subsection 4.3 using a simplified execution-time model for machine-level operations. In Subsection 4.4, a more refined model for operation execution times using probabilistic analysis is considered, and in Subsection 4.5 the effect of interaction between program segments is explored. The single-mode analysis of Section 4 introduces many concepts that are useful in Section 5. In Subsection 5.1, some of the challenges associated with mode-independent languages are considered by examining how programs written in a mode-independent manner can be executed on a mixed-mode machine. The single-mode analysis of Section 4 is expanded to find an efficient mixed-mode implementation in Subsection 5.2. Subsections 5.3 and 5.4 introduce a method to determine the minimum execution time in an SIMD/SPMD mixed-mode machine implementation through the use of multistage optimization techniques. Concluding remarks and areas for further research are summarized in Section 6.

2. MACHINE AND LANGUAGE MODEL

The mixed-mode machine model assumes a physically distributed memory. Thus, each processor is paired with a memory, forming a processing element (*PE*). It is assumed for mixed-mode machines that all PEs are in the same mode at any point in time (i.e., SIMD versus MIMD).

In the SIMD model used here, the control unit (*CU*) enqueues instructions to be performed on the PEs into an instruction queue. The PEs then fetch instructions from the instruction queue independently of CU operation. Thus, in SIMD mode, the CU can overlap its operations with those of the PEs.

One way of programming machines that are capable of mixed-mode parallelism is by using a *mode-independent language*, i.e., a language whose syntactic elements have interpretations under more than one mode of parallelism. An example of a mode-independent language is ELP, under development at Purdue University [31]. ELP provides constructs for SIMD, MIMD, and mixed-mode parallelism. The ELP syntax is based on C, and allows the programmer to specify the SIMD, MIMD, and SPMD modes of parallelism within a program. Although ELP contains specifiers for full MIMD mode processing, this paper focuses on SIMD and SPMD modes only. A goal of ELP is to provide uniformity with respect to the SIMD and SPMD modes of parallelism by having interpretations for the syntax within both of these modes that are identical in semantics. This is an important characteristic because it allows a data-parallel algorithm to be coded in a mode-independent manner, producing a data-parallel program for which (1) only SIMD code is generated, (2) only SPMD code is generated, or (3) execution-mode specifiers can be added to facilitate mixed-mode experimentation.

ELP associates with each variable defined in a program a *variable class*. A variable defined to be of class *mono* always has a single value with respect to all PEs, independent of execution mode, whereas a variable defined to be of class *poly* can have different values in each PE, independent of execution mode. Each mono variable has storage allocated for it on the CU and on all PEs. 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. For variables defined to be poly, each PE has its own copy with its own value, independent of execution mode.

In SIMD mode, operations on mono variables indicate 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 PEs. In SPMD mode, mono variables can be used to force `if`, `while`, `do`, and `for` statements on different PEs 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 PEs. All PEs must execute the same instructions, but not necessarily at the same time (as in SIMD mode). Thus, mono variables in ELP are guaranteed to have the same value spatially, but not temporally. This spatial congruency is enforced syntactically by the ELP compiler. Mono variables also permit other SPMD operations to be performed in an identical fashion across all PEs, such as having each PE access the same element of an array.

In the assumed model, when changing between SIMD mode and SPMD mode, only the source of the instructions (control) is changed between the CU and each PE's local memory, respectively. In both the SIMD and SPMD modes of execution, the same local memory and registers are used at each PE to store poly variables. Thus, mode changes in themselves do not cause a need for data transfers except for mono variables, the time for which is assumed to be relatively negligible. A more detailed description of the ELP language and compiler can be found in [31].

3. PARALLEL PERFORMANCE ISSUES

There are trade-offs that exist between the SIMD and MIMD modes of parallelism that explain why some sequences of instructions are better performed in one mode than in the other [6, 35]. Some of the advantages and disadvantages of each mode are summarized here.

Conditional statements in the synchronous execution of an SIMD program can introduce serialization. Consider an if-*A*-then-*B*-else-*C* statement. Let the conditional test *A* depend on PE data. In some PEs, *A* is true and in others false. Those PEs where *A* is false are disabled (masked off) during the execution of clause *B*. Once *B* has executed, the PEs where *A* is true are disabled and the PEs where *A* is false enabled. *C* is then executed. This serializes the execution of *B* and *C*. Conversely, in MIMD mode those PEs where *A* is true can execute *B* while the other PEs execute *C*. In MIMD mode, the maximum time to execute the if-then-else statement in a PE is approximately $T_A + \max(T_B, T_C)$, while in SIMD mode the time would be approximately $T_A + T_B + T_C$ (where a PE is idle for either T_B or T_C). Thus, in general, MIMD mode is more effective for executing conditional statements.

Another distinction between SIMD mode and MIMD mode pertains to synchronization overhead. In SIMD mode, the synchronization of program execution is implicit, because there is a single thread of control. However, when synchronization of program execution is required among PEs in MIMD mode, explicit synchronization mechanisms, such as semaphores or barriers, must be employed in the parallel program. Thus, synchronization costs are greater for MIMD mode. One benefit of implicit PE synchronization becomes apparent when inter-PE data transfers are needed. In SIMD mode, when one PE sends data to another PE, all enabled PEs send data. Therefore, the "send" and "receive" commands are implicitly synchronized. Because all enabled PEs follow the same single instruction stream, each PE knows from which PE the message has been received and for what use the message is intended. Conversely, MIMD mode programs are executed asynchronously among all PEs. As a result, the PEs must execute explicit synchronization and identification protocols for each inter-PE transfer. While the details of the inter-PE transfer protocol in both SIMD and MIMD mode are implementation-dependent, there is generally more overhead associated with MIMD mode inter-PE transfers. Like the synchronization overhead described above, this protocol overhead is a cost of the flexibility of programming in MIMD mode.

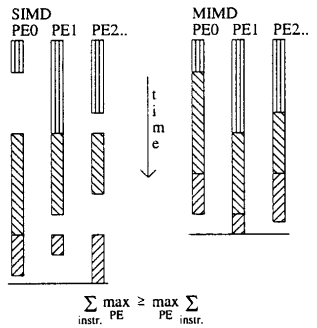

FIG. 1. Execution of variable-time instructions in SIMD and MIMD mode.

In SIMD mode, the CU can overlap its operations with those of the PEs. For example, the CU can perform the increment and compare operations on scalar-valued loop control variables, while the PEs execute the loop body. Furthermore, any operations common to all PEs, such as local array address calculations, can be performed in the CU while the PEs are performing other computations.

It is possible that the execution time of an instruction is data-dependent, taking a variable length of time to perform on each PE. Such *variable-time* instructions execute at least as efficiently in MIMD mode as in SIMD mode. This is illustrated in Fig. 1. In SIMD mode, PEs can execute the next instruction only after all PEs have completed the current instruction. Therefore, each instruction takes as long as it takes the PE that executes it most slowly. In MIMD mode, the PEs are not synchronized and each PE executes the next instruction independently. Let T_{ij} represent the time it takes instruction i to execute in PE j. Assume that T_{ij} in SIMD mode is equal to T_{ij} in MIMD model. The execution time in SIMD mode of a sequence of variable-time instructions can be expressed as $\sum_i \max_j(T_{ij})$, for all i in the sequence. The time to perform the same sequence of instructions in MIMD mode can be expressed in terms of T_{ij} as $\max_j(\sum_i T_{ij})$ (Fig. 1). Because $\max_j(\sum_i T_{ij}) \leq \sum_i \max_j(T_{ij})$, the time to execute the sequence of variable-time instructions in MIMD mode is less than or equal to the time to execute the same sequence of instructions in SIMD mode. Thus, because of this "sum of maxs" versus "max of sums" effect, MIMD mode is more appropriate for executing sequences of variable-time instructions.

The "sum of maxs" versus "max of sums" property is not limited to single instructions whose execution time is data-dependent. In mixed-mode, an entire block of in-

structions whose execution time varies on different PEs due to data-conditional statements and/or variable-time instructions may exhibit the same performance characteristics on a "macro" level if synchronization is required after the block. Consider a loop body with two blocks A and B, such that the first block is best executed in MIMD and the second block is best implemented in SIMD. Because all PEs must synchronize after block A, before any PE can go on to block B, an added synchronization cost is encountered during each iteration of the loop. Consequently, the time penalty for synchronizing with the other PEs may outweigh the benefit of implementing block A in SIMD; i.e., the overall execution time may possibly be reduced by implementing the loop entirely in MIMD mode [5]. The former case corresponds to the "sum of maxs" and the latter to the "max of sums."

From the discussion above it is evident that there are trade-offs between operating in SIMD mode and operating in MIMD mode. Although it is often clear in which mode a sequence of instructions should be implemented, this is not the case when counteracting trade-offs are involved. For example, a data-conditional clause may contain instructions that perform network transfers. Choosing the best mode of operation is not straightforward; i.e., conditional statements should be performed in MIMD mode while network transfers should be performed in SIMD mode. Studies examining the implementation of algorithms on mixed-mode machines have been performed (e.g., [4, 5, 9, 17, 21]). One long-term goal of these efforts has been to increase the understanding needed to develop automatic, static source-code-based determination of parallel modes for algorithm segments. In Sections 4 and 5, a methodology for analyzing data-parallel programs to do this is examined.

By employing the methodology developed in Sections 4 and 5, quantifications of all of the trade-offs discussed in this section can be incorporated in a straightforward manner, except for the "sum of maxs" versus "max of sums" trade-off, as well as the "macro" effect of this trade-off. Accurately modeling this trade-off is complex and replete with subtle nuances. In Section 4.4, a probabilistic basis for studying this aspect of SIMD/SPMD mode comparison is outlined.

4. SINGLE-MODE SELECTION

4.1. Overview of Single-Mode Selection

A methodology is presented here to estimate the execution time of a data-parallel algorithm, written in a mode-independent language, that can be implemented in either the purely SIMD or purely SPMD mode of parallelism. This framework can be used to select the optimal single mode for a mixed-mode parallel computer. Many of the concepts developed in this section are needed for the analysis in Section 5, where the use of both SIMD and SPMD within the same program is considered.

4.2. Assumptions and Definitions

Applications for this study are assumed to be *data-parallel* [24], i.e., the data for a program are distributed among the PEs, and the algorithm exhibits a high degree of uniformity across the data [26]. This is in contrast to *function (or control) parallelism* [38], where each PE executes a unique program.

Syntactic elements of the mode-independent language in which the algorithm is coded will be referred to as operations. *Operations* in a language represent the most explicit level at which program representation is identical for each mode of parallelism. Consider an operand fetch for a mono variable for use in a calculation that must be performed on the PEs. In SPMD mode, the operation is a simple fetch, because mono variables are stored on the PEs; however, in SIMD mode the mono variables are stored on the CU, and must be transferred to the PEs through the instruction queue or via a separate data bus. An operand fetch of a mono variable would therefore be considered a single operation for which execution-time information is known for both the SIMD and SPMD modes, even though different multiple machine level commands are required to perform the operation under each mode.

In general, the execution time of an operation may be fixed (and known) or data-dependent. For the case of data-dependent operations, a probabilistic model is introduced in Subsection 4.4 to estimate the expected execution time for a block of operations. It is assumed that the necessary statistical information can be estimated empirically and is known a priori.

Even if the target architecture for both SIMD and SPMD implementations is virtually identical, as is typically the case for a mixed-mode machine, execution times of the same operation under SIMD and SPMD modes may differ significantly. For example, the expected time required to perform the mono operand fetch described in a previous paragraph would generally be greater under SIMD mode than under SPMD mode.

To estimate the overall execution time of a parallel program, code is examined in terms of constructs and blocks. *Constructs* include *control constructs*, such as looping structures and function calls, and *data-conditional constructs*, such as `if-then-else` and `case` statements. For the analysis here, the set of permissible control constructs is restricted to `for` loops for which the number of iterations, Q, is assumed to be known at compile time. Furthermore, because programs under consideration exhibit characteristics that make them candidates for implementation in SIMD mode (in which all PEs iterate the same number of times) it is also assumed that all PEs executing a loop in SPMD mode iterate the same number of times. In a practical sense, the expected value for Q may be obtained directly from the program source, either directly as a constant in the program, or as information provided to the compiler by the programmer in

the form of a compiler directive. Alternatively, a sufficiently accurate value for Q might be obtained empirically by executing a prototype version of the parallel program on sample data sets. While time consuming, this empirical approach may be reasonable for production codes.

The set of data-conditional constructs is limited here to `if-then-else` constructs, where all PEs may or may not perform the same blocks of code, depending on the outcome of a condition test of local PE data. It is assumed that the necessary statistical information regarding the probability that a branch will be taken can be estimated empirically or is obtainable by compiler directives provided by the programmer.

Future work will examine relaxing the restrictions to include other constructs, e.g., `while` and `case` statements. It will also investigate methods for obtaining the statistical information assumed to be known here, and for eliminating the assumption that all PEs iterate the same number of times.

Code *blocks* are the parallel equivalent of the serial basic blocks described in [1]. Blocks of code are identified by their leading statements, called *leaders*. The first statement in a program is a leader, any statement that becomes the target of a conditional or unconditional branch at the machine code level is a leader, and any statement that follows a conditional branch at the machine code level is a leader. In the approach described here, an additional constraint is used in the determination of blocks: any statement requiring synchronization and any statement that follows a statement requiring a synchronization is a leader, and any statement requiring an inter-PE data transfer and any statement that follows an inter-PE data transfer is a leader. This is an important distinction from the serial definition of a basic block in [1], because synchronization points within an algorithm indicate when PEs are idle, waiting for one or more other PEs to arrive at the synchronization point.

Blocks consist of either pure scalar code or pure parallel code. *Scalar blocks* consist of instructions that, if performed in SIMD mode, would be executed on the CU (i.e., code that references only mono-valued variables). *Parallel blocks* consist of instructions to be performed on the PEs in SIMD mode (i.e., code that includes a reference to poly-valued variables). As an example, consider the execution of the loop `i=1` to `n` in SIMD mode, where both `i` and `n` are mono-valued variables, and where the loop body contains no references to mono-valued variables. One way to implement the loop is for the CU to perform an end-of-loop test, enqueue the appropriate SIMD instruction blocks to be broadcast to the PEs, perform an induction step, and branch back to the test. In this case, the PE instructions forming the body of the loop would be considered parallel blocks, while the increment and test code would each form a scalar block. Even if that portion of the program were to be executed in SPMD mode, the block definitions are the same as for the SIMD case.

4.3. Description of the Single-Mode Selection Technique

To determine the best single mode for a given program, the program is first transformed into a *flow-analysis tree*, which is a tree whose structure represents the scope levels within the algorithm. The flow-analysis tree is then used to create an *SIMD/SPMD trade-off tree*. For both trees, the leaf nodes of the tree represent parallel and scalar code blocks, and the nonleaf nodes correspond to control constructs and data-conditional constructs. The root node represents the scope of the entire program. Initially, only information about the execution times of the leaf nodes is assumed to be known. The proposed technique involves combining this known information to arrive at SIMD and SPMD execution times for the intermediate nodes. The optimal mode determined for the root node represents the "best" mode for the entire program. In this section, it is assumed that both the SIMD and SPMD execution times associated with each leaf node are known constants. In Subsection 4.4, methods are given for estimating SIMD and SPMD execution times for leaf nodes under the assumption that the execution times of the operations are random quantities with known probability distributions.

Figure 2 illustrates how a program is transformed into a preliminary flow-analysis tree. The program on the left side of the figure is composed of two elements, `block_a` and a `for` loop. Thus, the root node has two children, one for each of the elements. The `for` loop is composed of `block_b`, an `if` construct, and `block_f`, each of which is a child of the `for` node. For the `if` construct, the `then` clause is a single block (`block_c`), and is placed in the preliminary flow-analysis tree as a child of the `if` node. However, because the `else` clause is composed of `block_d` and `block_e`, an interior node representing the `else` clause is formed, whose children are the leaves `block_d` and `block_e` in Fig. 2. At each

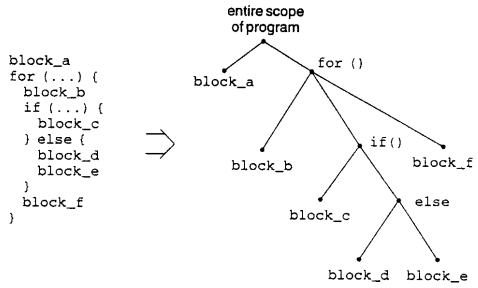

FIG. 2. Example preliminary flow-analysis tree.

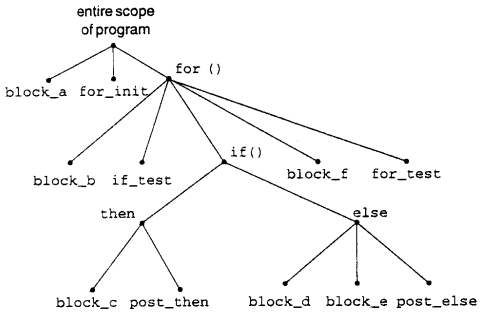

FIG. 3. Final flow-analysis tree after modification to include overhead for `for` and `if` constructs.

level in the tree, the sibling blocks are represented in the same order (left to right) as they appear in the program. This order dependence becomes important for the analysis of juxtaposed nodes and the use of SIMD/SPMD operations together, presented in Subsection 4.5 and Section 5, respectively.

To include the overhead required for `for` and `if` constructs, it is necessary to add nodes to the preliminary flow-analysis tree. This is illustrated for the previous example in Fig. 3.

Associated with each `for` loop is a block required for initializing induction variables, and another block required to execute the induction step for the loop, perform an end-of-loop test, and conditionally branch back to the top of the loop. Because the initialization block is executed only once, it is represented as a separate child of the root node in Fig. 3, labeled `for_init`. Because the increment, test, and conditional branch are executed during each iteration of the loop, they are represented by a single block as the last child of the `for` node in Fig. 3, labeled `for_test`.

Associated with each `if` construct is a conditional test (`if_test` in Fig. 3), performed before either the `then` clause or the `else` clause is executed. There is also special processing required at the end of the `then` clause (`post_then` in Fig. 3). In SPMD mode, this is an unconditional branch performed on each of the PEs for which the condition is true. In SIMD mode, this corresponds to disabling those PEs for which the condition is true and enabling those PEs for which the condition is false (with nested conditionals handled appropriately). Similarly, at the end of the `else` clause in SIMD mode, those PEs for which the condition is true must be re-enabled (`post_else` in Fig. 3). Because there is no corresponding operation required in SPMD, the cost for executing the `post_else` in SPMD mode is 0. Because the `then` clause of the `if` construct is now composed of two blocks, an interior node is added representing the `then` clause, with leaves `block_c` and `post_then` as children.

After the final flow-analysis tree is formed, the following steps are performed to convert it to an SIMD/SPMD trade-off tree, as illustrated in Fig. 4:

(1) For each leaf l of the tree, assign an ordered pair $(T_l^{\text{SIMD}}, T_l^{\text{SPMD}})$, which represents the times for executing the block associated with leaf l in SIMD mode and SPMD mode, respectively.

Steps 2 and 3 are then performed for each nonleaf node in the order of a depth-first traversal of the tree.

(2) For each nonleaf node d corresponding to a $d$ata-conditional construct, assign the ordered pair $(T_d^{\text{SIMD}}, T_d^{\text{SPMD}})$, which represents the times for executing the data-conditional construct at node d, including the time to execute all the children of d in SIMD mode and SPMD mode, respectively. To estimate the values of the ordered pair $(T_d^{\text{SIMD}}, T_d^{\text{SPMD}})$, consider how data conditionals are performed in SIMD and SPMD modes. In SIMD mode, if the conditional test for an `if-then-else` is true for all PEs, then only the PE instructions that belong to the `then` clause need to be broadcast to the PEs. Similarly, if the conditional test is false for all PEs, only the `else` clause needs to be broadcast. However, if the condition is true for some PEs and false for others, then the PE instructions for both the `then` and `else` clauses must be broadcast, effectively serializing the two clauses, as discussed in Section 3. Let p_{then} denote the probability that a PE executes the `then` clause, let P_{then} denote the probability that all PEs execute the `then` clause, and let P_{else} denote the probability that all PEs execute the `else` clause. For the special case when the outcomes of the conditional tests are mutually independent among N PEs, $P_{\text{then}} = (p_{\text{then}})^N$. In general, however, the assumption of mutual independence cannot be made, and therefore P_{then} and p_{then} must be determined separately. The values of P_{then}, P_{else}, and p_{then} can be estimated in a similar manner to the value of Q as discussed in Subsection 4.1 (e.g., empirical data and/or compiler directives). Let T_u^{SIMD} be the execution time in SIMD associated with node u, and

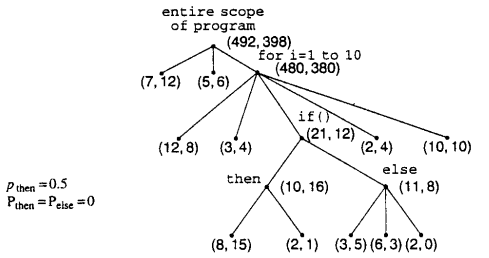

FIG. 4. SIMD/SPMD trade-off tree for the flow-analysis tree in Fig. 3.

let

$$T_{\text{then}}^{\text{SIMD}} = \sum_{\substack{\text{all children } u \\ \text{of then clause}}} T_u^{\text{SIMD}}$$

and

$$T_{\text{else}}^{\text{SIMD}} = \sum_{\substack{\text{all children } u \\ \text{of else clause}}} T_u^{\text{SIMD}}.$$

Then the expected time required to perform the `if-then-else` in SIMD mode can be estimated by

$$\begin{aligned} T_d^{\text{SIMD}} &= P_{\text{then}} T_{\text{then}}^{\text{SIMD}} + P_{\text{else}} T_{\text{else}}^{\text{SIMD}} \\ &\quad + (1 - P_{\text{then}} - P_{\text{else}})(T_{\text{then}}^{\text{SIMD}} + T_{\text{else}}^{\text{SIMD}}) \\ &= (1 - P_{\text{else}}) T_{\text{then}}^{\text{SIMD}} + (1 - P_{\text{then}}) T_{\text{else}}^{\text{SIMD}}. \end{aligned}$$

In contrast to SIMD mode, in SPMD mode each PE independently follows its own control path through the program. The `then` clause is executed on those PEs where the condition is true, and the `else` clause is executed on those PEs where the condition is false. Let T_u^{SPMD} be the execution time in SPMD associated with node u, and let

$$T_{\text{then}}^{\text{SPMD}} = \sum_{\substack{\text{all children } u \\ \text{of then clause}}} T_u^{\text{SPMD}}$$

and

$$T_{\text{else}}^{\text{SPMD}} = \sum_{\substack{\text{all children } u \\ \text{of else clause}}} T_u^{\text{SPMD}}.$$

Then the expected time to perform the conditional construct in SPMD mode is

$$T_d^{\text{SPMD}} = p_{\text{then}} T_{\text{then}}^{\text{SPMD}} + (1 - p_{\text{then}}) T_{\text{else}}^{\text{SPMD}}.$$

The estimates for T_d^{SIMD} and T_d^{SPMD} derived above are expected execution times and will not necessarily be the actual (observed) times for any particular execution. To clarify the meaning and use of the formulas for T_d^{SIMD} and T_d^{SPMD}, consider a simple example. Assume there is a sequence of m data conditionals with expected execution times $(T_{d_1}^{\text{SIMD}}, T_{d_1}^{\text{SPMD}})$, $(T_{d_2}^{\text{SIMD}}, T_{d_2}^{\text{SPMD}})$, $\cdots$ $(T_{d_m}^{\text{SIMD}}, T_{d_m}^{\text{SPMD}})$. To simplify the illustration, assume that $p_{\text{then}} = 0.5$ for all m conditionals, that $T_{\text{then}}^{\text{SIMD}} = T_{\text{else}}^{\text{SIMD}} = T_{\text{then}}^{\text{SPMD}} = T_{\text{else}}^{\text{SPMD}} = T$, and that the conditional tests are mutually independent, which implies that $P_{\text{then}} = (p_{\text{then}})^N$ and $P_{\text{else}} = (1 - p_{\text{then}})^N$. Assuming N is moderately large, the formula for the expected execution time in SIMD mode yields $T_{d_i}^{\text{SIMD}} \cong 2T$, $i \in \{1 \cdots m\}$. The formula for the expected execution time in SPMD mode yields $T_{d_i}^{\text{SPMD}} = T$, $i \in \{1 \cdots m\}$. The expected execution time for executing all m conditionals in SIMD mode is therefore approximately $2mT$, while the expected execution time for executing all m conditionals in SPMD mode is mT.

(3) For each nonleaf node c corresponding to a control construct, assign the ordered pair $(T_c^{\text{SIMD}}, T_c^{\text{SPMD}})$, which represents the times for executing the control construct at node c, including the time to execute all the children of c in SIMD mode and SPMD mode, respectively. In SPMD mode, the control construct is performed entirely on the PEs. In SIMD mode, the control steps are performed on the CU, and then the PE instructions that form the loop body are placed in the instruction queue to be broadcast to the PEs. For SIMD, significant improvement in execution time can be obtained by overlapping the execution of operations on the CU and the PEs, and the need for adding this to the existing model in the future is discussed in Subsection 4.5. Disregarding for the moment the effects of CU/PE overlap, the values of the ordered pair $(T_c^{\text{SIMD}}, T_c^{\text{SPMD}})$ can be estimated by

$$T_c^{\text{SIMD}} = Q\left(\sum_{\substack{\text{all children} \\ u \text{ of } c}} T_u^{\text{SIMD}}\right)$$

$$T_c^{\text{SPMD}} = Q\left(\sum_{\substack{\text{all children} \\ u \text{ of } c}} T_u^{\text{SPMD}}\right),$$

where Q is the number of iterations in the construct and is assumed to be known, as discussed in Subsection 4.2.

(4) For the node representing the root, assign the ordered pair $(T_{\text{root}}^{\text{SIMD}}, T_{\text{root}}^{\text{SPMD}})$, which represents the times for executing the entire program in SIMD mode and SPMD mode, respectively, where

$$T_{\text{root}}^{\text{SIMD}} = \sum_{\substack{\text{all children} \\ u \text{ of root}}} T_u^{\text{SIMD}}$$

and

$$T_{\text{root}}^{\text{SPMD}} = \sum_{\substack{\text{all children} \\ u \text{ of root}}} T_u^{\text{SPMD}}.$$

If $T_{\text{root}}^{\text{SPMD}} \leq T_{\text{root}}^{\text{SIMD}}$, then implement the algorithm in SPMD mode, else implement it in SIMD mode.

As an example, consider the SIMD/SPMD trade-off tree in Fig. 4, which corresponds to the flow-analysis tree in Fig. 3. In the tree, the ordered pairs $(T_l^{\text{SIMD}}, T_l^{\text{SPMD}})$ are assigned to each of the leaf nodes. For the example, let $Q = 10$, $p_{\text{then}} = 0.5$, and let $P_{\text{then}} = P_{\text{else}} = 0$. The analysis algorithm performs a depth-first traversal beginning at the root node. When the algorithm considers the `then` clause, it assigns the ordered pair $(T_{\text{then}}^{\text{SIMD}}, T_{\text{then}}^{\text{SPMD}}) = (8 + 2, 15 + 1) = (10, 16)$. Similarly, for the `else` clause,

$(T_{\text{else}}^{\text{SIMD}}, T_{\text{else}}^{\text{SPMD}}) = (3 + 6 + 2, 5 + 3 + 0) = (11, 8)$. Then the algorithm considers the `if` node, and assigns the ordered pair $(T_d^{\text{SIMD}}, T_d^{\text{SPMD}}) = (10 + 11, 0.5 \times 16 + 0.5 \times 8) = (21, 12)$. At the `for` node, the algorithm assigns the ordered pair $(T_c^{\text{SIMD}}, T_c^{\text{SPMD}}) = (10 \times (12 + 3 + 21 + 2 + 10), 10 \times (8 + 4 + 12 + 4 + 10)) = (480, 380)$. For the root node, the ordered pair $(7 + 5 + 480, 12 + 6 + 380) = (492, 398)$ is assigned, indicating that SPMD mode is best suited for this program.

4.4. A Probabilistic Model for Data-Dependent Operation Execution Times

The technique of Subsection 4.3 for estimating the execution time of a program operating in either SIMD mode or SPMD mode assumes that the execution times T_l^{SIMD} and T_l^{SPMD} are known constants for each leaf node l in the flow-analysis tree. While the leaf blocks do not contain conditional or control statements, times for these blocks may be data-dependent if the block includes instructions whose execution times are data-dependent (e.g., floating point addition). In this subsection, a probabilistic model to account for the uncertainty in the execution times of the blocks of code associated with the leaf nodes of the flow-analysis tree is considered.

The model presented here can be used as a basis for describing the general behavior of code segments whose execution time is data-dependent. The model does not attempt to completely describe program behavior. Instead, it seeks to estimate expected execution times for individual code segments. Of particular interest is the difference between the expected execution time of a parallel code block across all PEs and the expected execution time of the same block for the PE which takes the most time to complete execution of the block. This difference represents a cost associated with performing a synchronization across PEs.

Let k_l denote the number of operations in the block of code associated with the leaf node l. Label the operations in the block of code associated with the leaf node l as $0, 1, \ldots, k_l - 1$. For each leaf node l, define an array of continuous random variables denoted by X_{ij}^l, $i \in \{0, 1, \ldots, k_l - 1\}$, $j \in \{0, 1, \ldots, N - 1\}$, where N is the number of PEs executing that block. The value of the random variable X_{ij}^l corresponds to the execution time of operation i executing on PE j. In a mixed-mode machine, where all of the PEs are of the same architecture, it may be reasonable to assume that the probability distribution of the random variable X_{ij}^l is the same, regardless of the mode of execution of operation i. Examples of operations whose times will differ include statements that require the accessing of mono variables, for reasons discussed earlier, and inter-PE transfers, where the software overhead for SPMD is much greater than for SIMD. To ease the notational burden, this possible distinction in the distribution of X_{ij}^l for SIMD and SPMD is not explicitly indicated, but implied.

The *expected value* of the random variable X_{ij}^l, i.e., $E[X_{ij}^l]$, is assumed to exist and is denoted by μ_{ij}^l. The *variance* of the random variable X_{ij}^l, i.e., $E[(X_{ij}^l - \mu_{ij}^l)^2]$, is assumed to exist and is denoted by $(\sigma_{ij}^l)^2$. Recall that the expected value of a function of a continuous random variable X, say $g(X)$, is defined by $E[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x) dx$, where $f_X(x)$ is the probability density function for the random variable X [32].

For the purpose of this paper, it is assumed that the random variables X_{ij}^l are mutually independent for all $i \in \{0, 1, \ldots, k_l - 1\}, j \in \{0, 1, \ldots, N - 1\}$. Furthermore, it is assumed that for each $i \in \{0, 1, \ldots, k_l - 1\}$, the random variables X_{ij}^l are independent and identically distributed for all $j \in \{0, 1, \ldots, N - 1\}$. Thus, for each value of i, μ_{ij}^l will be denoted as μ_i^l and $(\sigma_{ij}^l)^2$ will be denoted as $(\sigma_i^l)^2$.

The random variable for the execution time associated with implementing a block of k_l operations in SIMD mode is defined by the following transformation (the "sum of the maxs" referred to in Section 3):

$$X_l^{\text{SIMD}} = \sum_{i=0}^{k_l - 1} \left(\max_{j \in \{0,1,\ldots,N-1\}} \{X_{ij}^l\} \right).$$

The random variable for the execution time associated with implementing a block of k_l operations in SPMD mode is defined by the following transformation (the "max of the sums" referred to in Section 3):

$$X_l^{\text{SPMD}} = \max_{j \in \{0,1,\ldots,N-1\}} \left\{ \sum_{i=0}^{k_l - 1} X_{ij}^l \right\}.$$

The above formula assumes that the PEs are synchronized both when they enter and when they exit the block associated with leaf node l. If the block is preceded and/or followed by a SPMD block that does not require synchronization, then this formula represents a worst case estimate.

If the probability distribution for each X_{ij}^l is assumed to be known, then it is possible (although tedious) to determine the exact probability distribution for both X_l^{SIMD} and X_l^{SPMD}. For the purposes of the present paper, only bounds for the expected values of these random variables will be determined. Upper bounds for $E[X_l^{\text{SIMD}}]$ and $E[X_l^{\text{SPMD}}]$ are derived next.

The expected value of X_l^{SIMD} is given by:

$$E[X_l^{\text{SIMD}}] = E\left[\sum_{i=0}^{k_l - 1} \max_{j \in \{0,1,\ldots,N-1\}} \{X_{ij}^l\} \right].$$

By the linearity of the $E[\cdot]$ operation, it follows that

$$E[X_l^{\text{SIMD}}] = \sum_{i=0}^{k_l - 1} E\left[\max_{j \in \{0,1,\ldots,N-1\}} \{X_{ij}^l\} \right]. \tag{1}$$

Because for each $i \in \{0, 1, \ldots, k_l - 1\}$ the random variables X_{ij}^l are independent and identically distributed with mean μ_i^l and variance $(\sigma_i^l)^2$ for all $j \in \{0, 1, \ldots, N-1\}$, a standard result from order statistics [13] can be applied to bound each term in the summation of Eq. 1. In particular,

$$E[\max_{j\in\{0,1,\ldots,N-1\}} \{X_{ij}^l\}] \leq \mu_i^l + \sigma_i^l \frac{(N-1)}{(2N-1)^{1/2}}.$$

Thus, it follows that

$$E[X_l^{\text{SIMD}}] \leq \sum_{i=0}^{k_l-1} \mu_i^l + \frac{(N-1)}{(2N-1)^{1/2}} \left(\sum_{i=0}^{k_l-1} \sigma_i^l\right). \quad (2)$$

Next, an upper bound for $E[X_l^{\text{SPMD}}]$ is derived. Define $S_j^l = \sum_{i=0}^{k_l-1} X_{ij}^l$, for each $j \in \{0, 1, \ldots, N-1\}$. Thus, the random variable X_l^{SPMD} can be expressed as

$$X_l^{\text{SPMD}} = \max_{j\in\{0,1,\ldots,N-1\}} \{S_j^l\}.$$

Because of the independence assumption, the expected value and variance of the random variables S_j^l are given by $\sum_{i=0}^{k_l-1} \mu_i^l$ and $\sum_{i=0}^{k_l-1} (\sigma_i^l)^2$, respectively. The upper bound result from [13] can be applied to bound $E[X_l^{\text{SPMD}}]$, as follows:

$$E[X_l^{\text{SPMD}}] \leq \sum_{i=0}^{k_l-1} \mu_i^l + \frac{(N-1)}{(2N-1)^{1/2}} \left(\sum_{i=0}^{k_l-1} (\sigma_i^l)^2\right)^{1/2}. \quad (3)$$

The upper bounds for $E[X_l^{\text{SIMD}}]$ and $E[X_l^{\text{SPMD}}]$ could be used to approximate the execution times T_l^{SIMD} and T_l^{SPMD}, assumed to be given constants in the previous subsection. Determining the actual distributions for the random variables X_l^{SIMD} and X_l^{SPMD} could give a better indication of what the fundamental trade-offs are between the SIMD and SPMD modes. For instance, the exact value for the mean and variance of X_l^{SIMD} and X_l^{SPMD} could be computed. In certain applications, a moderate mean execution time with an associated small variance may be preferred over a smaller mean execution time with a relatively large variance.

The above difference between SIMD and SPMD involves just one aspect of the relationship between these two modes of parallelism. There are other trade-offs, as discussed in Section 3, that must also be included in a determination of the best mode to use.

4.5. *Effects of Block Juxtaposition*

The total execution time for all children of a nonleaf node is estimated in Subsection 4.3 as the sum of the associated execution times. This estimate can be sharpened by considering the effects of juxtaposing blocks of code in either the SIMD or SPMD mode.

Consider first the case of estimating the total SIMD execution time for several sibling blocks. One factor that can have a significant effect is the overlapped execution of operations on the CU and PEs while in SIMD mode. A detailed analysis and explanation of CU/PE overlap, such as that presented in [28], is beyond the scope of this paper; however, the effect of CU/PE overlap can be accounted for in the present framework by using a simplified model. Consider a sequence of m blocks of code, say $B_j, B_{j+1}, \cdots B_{j+m}$, where each block is a pure parallel code block or pure scalar code block. Let $Ov(B_j, B_{j+1}, \cdots B_{j+m})$, denote the amount of execution time overlap among the blocks. Thus, the total execution time for a sequence of m such blocks in SIMD mode is given by

$$T_{j,\ldots,j+m} = \sum_{i=j}^{j+m} T_i - \text{Ov}(B_j, \ldots, B_{j+m}),$$

which represents the difference between the sum of the expected execution times of each of the blocks and the CU/PE overlap as a result of the juxtaposition of those blocks in SIMD mode.

As an example, consider Fig. 5, where two parallel blocks, a and c, are to be performed on the PEs, and block b is to be performed on the CU in SIMD mode. Blocks a and b require a total of 5 time units; however, the CU and the PEs overlap execution for 1 time unit. Similarly, although blocks b and c require a combined total of 6 time units, they also overlap execution for 1 time unit. For this example, Ov(a,b,c) = 2, $T_{\text{a,b,c}} = T_{\text{a}} + T_{\text{b}} + T_{\text{c}} - \text{Ov(a, b, c)} = (2 + 3 + 3 - 2) = 6$.

In SPMD mode, PEs may need to perform some type of synchronization at the end of a block. The cost of synchronizing is related to the amount of processing performed since the previous synchronization.

For this paper, $T_{j,\ldots,j+m}^{\text{SIMD}}$ and $T_{j,\ldots,j+m}^{\text{SPMD}}$ are approximated by the sum of the (known or estimated) individual block execution times, i.e., $T_{j,\ldots,j+m}^{\text{SIMD}} = \sum_{i=j}^{j+m} T_i^{\text{SPMD}}$ and $T_{j,\ldots,j+m}^{\text{SPMD}} = \sum_{i=j}^{j+m} T_i^{\text{SPMD}}$. Extending the probabilistic model of Subsection 4.4 to describe the probability distributions of the execution times of all types of leaf and nonleaf nodes is possible but not straightforward.

For example, consider the case of a sequence of two or

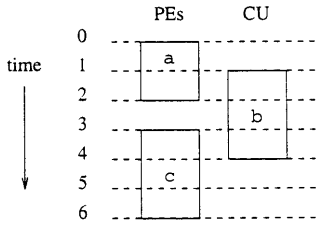

FIG. 5. Overlapped execution of CU and PE instructions.

more nodes to be implemented in SPMD mode. If no synchronization is required between nodes, then a smaller than expected execution time may result, because of the effect of the macro-level "sum of maxs" versus "max of sums" property described in Section 3. For a practical implementation, an extension of the concepts presented in Section 4.4 is needed.

4.6. Summary of Single-Mode Model

In this section, a model for determining the single mode of parallelism for which execution time is minimized has been described. Beginning with a program written in a mode-independent language, and given basic information about the execution time of the operations that are included in the language, useful execution time information for blocks within the program can be estimated. Employing a set of rules, block information is used to determine how fast a program will execute under either the SIMD or SPMD mode of parallel processing. This comparison technique can be used at compile time to determine which single-mode implementation on a mixed-mode machine would result in the minimum execution time.

5. MIXED-MODE ANALYSIS

5.1. Assumptions and Definitions

In mixed-mode systems, it is possible for some portions of a single parallel algorithm to be implemented in SIMD mode, and other portions of the same program to be executed in SPMD mode. The programmer can select the mode of parallelism best suited for each segment of the program. The analysis of single-mode parallel programs developed in the previous section is expanded here for the implementation of data-parallel programs in SIMD/SPMD mixed-mode environments. In this section, an algorithm that may use both SIMD and SPMD modes will be referred to as a *mixed algorithm*.

The execution time estimation technique for mixed-algorithm programs is a generalization of the single-mode case. For the analysis in this section, several simplifying assumptions are made on the selection of modes within a program.

First, leaf blocks are assumed to be implemented completely in either SIMD mode or SPMD mode. Given that block boundaries are defined by control-flow and data-conditional constructs, synchronizations, inter-PE data transfers, and CU operations that can be overlapped in SIMD mode, there is no benefit in changing modes within a leaf block. Mode changes are therefore allowed only at interblock boundaries.

Another assumption is that, if a block is to be executed more than once, i.e., as part of a looping construct, the mode of parallelism for that block is the same for all iterations. Because there are no branches or targets of branches within a block, and no inter-PE transfers or synchronization points within a block, there is no perceived advantage for the same block to execute in different modes from iteration to iteration; i.e., if a given mode is better for one instance of a block, it should be better for all instances.

Additionally, all blocks that comprise each data-conditional construct are to be implemented in the same mode of parallelism; i.e., for each `if` construct, all the blocks within the construct are either implemented in SIMD mode, or they are all implemented in SPMD mode. This assumption is necessary because mode changes within data-conditional constructs can translate into operations that cannot be implemented without excessive execution-time overhead. For example, consider an SPMD data-conditional construct with an embedded SIMD block. Because each PE independently performs a test to determine whether to perform the `then` clause or the `else` clause of a data-conditional construct, the CU cannot know which PEs are performing each clause. The CU therefore cannot know when all the appropriate PEs have arrived at the SIMD block, because some PEs may never execute the clause that contains the SIMD block. The situation can become worse when conditionals are nested. This type of problem led to the operational constraint of the mixed-mode model, given in Section 2, that all PEs must be in the same mode at a given point in time.

Finally, it is assumed that each iteration of a loop must begin and end execution in the same mode of parallelism. Consider, for example, a sequence of blocks that comprise the body of a loop, such that the first block is implemented in SIMD mode and the last block is implemented in SPMD mode. Between each successive iteration of the loop, a mode switch from SPMD to SIMD is required to allow the PEs to perform the first block of the next iteration. Because the last block does not end in the same mode as the first block, a mode switch is added at the beginning of the loop body.

5.2. Optimal Selection of Modes for Mixed Algorithms

To perform the execution time analysis and optimization for a mixed algorithm, an SIMD/SPMD trade-off tree is constructed, where, as before, each block in the program is a leaf node and the data-conditional and control constructs form the interior nodes, with the root node representing the scope of the entire program. For mixed algorithms, the mode of parallelism may be changed between adjacent sibling nodes, with an appropriate time cost. These mode changes are not nodes and are not represented in the SIMD/SPMD trade-off tree.

For the following discussion, let the ordered pair $(T^{\text{SIMD}}_{\text{mixed-}n}, T^{\text{SPMD}}_{\text{mixed-}n})$ represent the minimum mixed-algorithm execution time estimate for a nonleaf node n, where n is not the root for the entire program, i.e., for a subtree with root n that begins and terminates execution in one of SIMD or SPMD, respectively, but may switch modes zero or more times during execution. The values

for $T^{SIMD}_{mixed\text{-}n}$ and $T^{SPMD}_{mixed\text{-}n}$ include the time required for any mode switches that are performed. In the SIMD/SPMD trade-off tree, interior nodes can represent `if` constructs, `then` and `else` clauses, and `for` constructs. Recall that for nonleaf nodes corresponding to descendents of data-conditional constructs, mode changes are disallowed. Therefore, for nodes corresponding to `if`, `then`, and `else` nodes, $T^{SIMD}_{mixed\text{-}n}$ and $T^{SPMD}_{mixed\text{-}n}$ represent the estimated execution time for that node purely in SIMD or purely in SPMD, which can be determined using the techniques given in Section 4. Recall that for a sequence of nodes that are children of a `for` construct, the modes of the nodes can differ. However, the first and last nodes must be implemented in the same mode of parallelism or a mode switch must be added before the first node. Therefore, the mode of the `for` subtree is defined to be that of the last node that is a child of that `for` node. Thus, the only subtree that does not necessarily begin and terminate in the same mode is the node corresponding to the root of the entire program.

There is a cost C^{SIMD} associated with switching to SIMD mode, and a cost C^{SPMD} associated with switching to SPMD mode. The values of C^{SIMD} and C^{SPMD} are assumed to be known constants. The value of C^{SIMD} is determined by the execution time of the mode switching hardware and software, and does not include the time it takes between the first PE reaching the mode change synchronization point and the last PE reaching that point. Although the time to synchronize PEs is generally not a fixed constant because it depends on the amount of processing performed since the last synchronization, for the purposes of this framework a "typical" synchronization time is assumed to be included in the cost C^{SIMD} associated with switching to SIMD mode. Explicitly including these times into the analysis relates to the discussion in Subsection 4.5 and is beyond the scope of this paper.

For the case of a heterogeneous suite of parallel machines, C^{SIMD} and C^{SPMD} will typically not be constants and will generally be greater than for a single mixed-mode machine, because data may have to be moved between machines. The application of this technique under the relaxed assumption of nonconstant mode-switching costs for a heterogeneous suite of parallel machines is currently under study [40].

The methodology of Subsection 4.3 is adapted below for analyzing mixed algorithms. The following steps are performed to determine the execution times for all nodes:

(1) For each leaf l of the tree, assign an ordered pair (T^{SIMD}_l, T^{SPMD}_l), where T^{SIMD}_l is the SIMD execution time estimate for block l, and T^{SPMD}_l is the SPMD execution time estimate for block l, determined as in the single-mode analysis presented in Section 4.

Steps 2 and 3 are performed for each nonleaf node in the order of a depth-first traversal of the tree:

(2) For each nonleaf node d corresponding to a data-conditional construct, assign an ordered pair $(T^{SIMD}_{mixed\text{-}d}, T^{SPMD}_{mixed\text{-}d})$, which represents the times for executing the data-conditional construct at node d, including the time to execute all the children of d. Analogous to the single-mode case presented in Subsection 4.3, an estimate for $(T^{SIMD}_{mixed\text{-}d}, T^{SPMD}_{mixed\text{-}d})$, is given by

$$T^{SIMD}_{mixed\text{-}d} = (1 - P_{else})T^{SIMD}_{mixed\text{-}then} + (1 - P_{then})T^{SIMD}_{mixed\text{-}else}$$

$$T^{SPMD}_{mixed\text{-}d} = p_{then}T^{SPMD}_{mixed\text{-}then} + (1 - p_{then})T^{SPMD}_{mixed\text{-}else}.$$

Because of the single execution-mode constraint assumed for data-conditional constructs, $T^{SIMD}_{mixed\text{-}then} = T^{SIMD}_{then}$, $T^{SPMD}_{mixed\text{-}then} = T^{SPMD}_{then}$, $T^{SIMD}_{mixed\text{-}else} = T^{SIMD}_{else}$, and $T^{SPMD}_{mixed\text{-}else} = T^{SPMD}_{else}$, and the above equations reduce to the single-mode case. The formula used for $T^{SPMD}_{mixed\text{-}d}$ is the expected time for each PE to execute the data-conditional construct at node d, and not necessarily the maximum time taken over all PEs for a given execution. Thus, the value of $T^{SPMD}_{mixed\text{-}d}$ does not account for the time to synchronize the PEs for the case where the node following node d is executed in SIMD mode.

(3) Case (a). For each nonleaf node c corresponding to a control construct that is not in a subtree with a data-conditional construct as its root, assign an ordered pair $(T^{SIMD}_{mixed\text{-}c}, T^{SPMD}_{mixed\text{-}c})$, which represents the times for executing the control construct at node c, beginning and ending in SIMD and SPMD modes, respectively, including the time to execute all the children of c. Let the ordered pair $(T^{SIMD}_{mixed\text{-}iteration}, T^{SPMD}_{mixed\text{-}iteration})$ represent the minimum mixed-algorithm execution time estimate for a single iteration of the loop corresponding to node c. Because the last block in the loop is the `for_test`, and because the first and the last block must be in the same mode of parallelism, $T^{SIMD}_{mixed\text{-}iteration}$ corresponds to performing the `for_test` in SIMD on the CU, and $T^{SPMD}_{mixed\text{-}iteration}$ corresponds to performing the `for_test` in SPMD on the PEs. Recall that Q is the number of iterations of the loop to be executed. Then $(T^{SIMD}_{mixed\text{-}c}, T^{SPMD}_{mixed\text{-}c})$ is given by

$$T^{SIMD}_{mixed\text{-}c} = Q \times T^{SIMD}_{mixed\text{-}iteration}$$

$$T^{SPMD}_{mixed\text{-}c} = Q \times T^{SPMD}_{mixed\text{-}iteration}.$$

Recall that if the first node of a loop body and the `for_test` are in different modes, a mode switch is inserted before the first node in the loop body. This may lead to an unneeded mode switch for the first iteration. This situation is described in detail below.

Case (b). If node c is the descendent of a node corresponding to an `if` construct, then only pure SIMD and pure SPMD implementations are considered, and the equations for the single-mode analysis methodology are used instead.

(4) For the node representing the root, assign the ordered quadruple $(T^{SIMD/SIMD}_{root}, T^{SIMD/SPMD}_{root}, T^{SPMD/SIMD}_{root}, T^{SPMD/SPMD}_{root})$, where $T^{X/Y}_{root}$ corresponds to the minimum mixed-algorithm execution time required to perform all the children of the root node, beginning in mode X and

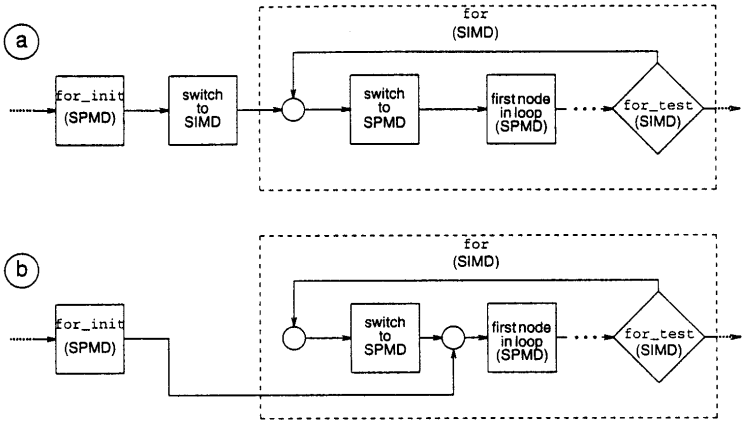

FIG. 6. Modification of `for` loop to avoid unnecessary mode switches for the first iteration: (a) with unneeded mode switch; (b) without unneeded mode switch.

ending in mode Y (where zero or more mode switches can occur between X and Y). Thus, it is possible for the root node to avoid a final mode change. The minimum quadruple value then represents the best mixed-mode implementation.

As the SIMD/SPMD trade-off tree is traversed, the deepest levels of the tree are combined by employing the above steps. As the analysis works its way up the tree, higher levels are combined, until only the root is represented. Then the parallel mode for each segment of the program can be assigned.

One effect that must be considered when traversing the tree is the possibility of adding unnecessary mode switches when going from the `for_init` block to the first child in the loop body of a `for` node. For example, consider a `for` construct such that the `for_init` block is implemented in SPMD, the first child in the loop body for the `for` node is implemented in SPMD, and the `for_test` block (the last child of the `for` node) is implemented in SIMD, as illustrated in Fig. 6a. Because the last and first nodes are performed one after the other when iterating over the loop, a mode switch from SIMD to SPMD is required before executing the first node. Additionally, by the definition of a `for` loop node, the mode of the `for` node is the same as that of the `for_test` in the loop (as stated in Subsection 5.1). Thus, a mode switch from the SPMD `for_init` node to the SIMD `for` node is inserted before the first iteration of the loop body. However, immediately after the execution of the `for_init`, and before the first iteration, the parallel system is already in SPMD mode, which is the mode of parallelism required for the first node in the loop body. By detecting this case, the mode switch from SPMD to SIMD and from SIMD back to SPMD can be avoided for the first iteration, as illustrated in Fig. 6b. Thus, it is beneficial for the analysis to recognize this case and to remove mode switches that are not needed for the first iteration of the loop from the time-cost estimate.

One element of the analysis not addressed, which should be included in a practical implementation of the techniques presented here, is the effect of implementing a sequence of two or more nodes in SPMD mode. Recall from Subsection 4.5, if an SPMD node is directly followed by another SPMD node with no intervening synchronization, then a reduced execution time may result, due to the macrolevel "sum of maxs" versus "max of sums" effect described in Section 3. This effect is particularly applicable to the estimated execution time of `for` constructs. Thus, for a practical implementation, an extension of the concepts presented in Subsection 4.4 is needed, where such adjacent SPMD nodes are treated as a single node for the purposes of the type of analysis discussed in that subsection.

5.3. Computational Aspects of Optimal Selection of Modes for Mixed Algorithms

Exhaustively testing all possible implementations to find the minimum mixed-algorithm execution time for a node with m children would require testing 2^m combinations. For nodes with many children, this approach is impractical; therefore, an efficient method of finding the minimum execution time is needed when m becomes large.

An efficient way of computing $(T^{\text{SIMD}}_{\text{mixed-}n}, T^{\text{SPMD}}_{\text{mixed-}n})$ is to transform the subtree rooted at n into a *multistage optimization* problem. In multistage optimization problems, proceeding to stage $j + 1$ is possible only by passing through stage j. There is a cost associated with proceeding from stage j to stage $j + 1$, depending on the initial

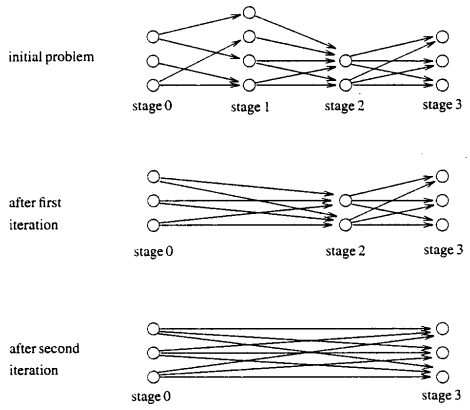

FIG. 7. Example of a general multistage optimization problem and the aggregate structure of the intermediate and final solution graphs.

state (at stage j) and the final state (at stage $j + 1$). In a multistage optimization graph, each state in each stage is represented by a vertex, and edges connecting vertices in stage j to vertices in stage $j + 1$ indicate valid transitions, each with an associated cost. The goal of a multistage optimization problem is to find the minimum cost path between the initial and final stages, e.g., see [2, 10].

Multistage optimization problems can be solved using Moore's algorithm [30] (a variant of Dijkstra's algorithm [14]) in $s - 2$ iterations for an s-stage problem. In each iteration, two successive stages of the multistage optimization graph are reduced to a single stage, so that at the end of $s - 2$ iterations only the initial and final stages remain, with connecting edges indicating the minimum cost from each of the states in the initial stage to each of the states in the final stage.

An example of a general multistage optimization problem and its solution is illustrated in Fig. 7. Initially, there are four stages, numbered from 0 to 3. In the first iteration, for each vertex g in stage 0, a minimum path is found from g to each vertex h in stage 2, passing through stage 1. Stage 1 is then removed from the graph, and new edges are drawn from vertices in stage 0 to vertices in stage 2, indicating the minimum cost to proceed from stage 0 to stage 2 for each possible (g, h) pair. The multistage optimization algorithm is applied again, and the problem is reduced to two stages, indicating the minimum cost to proceed from each initial state to each final state. By recording the minimum paths selected during each iteration of the algorithm, the path through the entire multistage problem resulting in the minimum cost for each initial/final pair is obtained. The correctness of the optimization algorithm is based on the principle that all subpaths along an optimal (i.e., shortest) path must themselves be optimal.

To find the minimum execution time for all the children of a node in the SIMD/SPMD trade-off tree, let a stage and the edges exiting that stage in a multistage optimization graph correspond to a single child node. Let each mode of parallelism represent a separate state within each stage. Let the cost associated with each edge correspond to the cost of performing that node in SIMD mode or SPMD mode. Then, before the representation of each node in the multistage graph, include a stage and associated edges representing the cost of a possible mode switch between nodes.

As an example, consider the transformation illustrated in Fig. 8a. To find the minimum mixed algorithm execution time for a sequence of sibling nodes, each node is modeled as three stages of a multistage graph (numbered 0 to 2 in Fig. 8a) and then joined together to form a multistage optimization graph for the entire sequence of nodes. In each stage, the upper vertex represents SIMD mode, while the lower vertex represents SPMD mode. For stage 0 and its associated edges, a possible mode change is represented. The edge from the upper vertex in the stage 0 to the lower vertex in stage 1 is labeled with the cost of switching from SIMD to SPMD mode. Similarly, the cost of switching from SPMD to SIMD is indicated as the label for the edge from the lower vertex in stage 0 to the upper vertex in stage 1. There is zero cost for remaining in the same mode of parallelism. The edges from stage 1 to stage 2 in Fig. 8a represent the cost of executing the node in each mode. Figure 8b illustrates transforming an SIMD/SPMD trade-off tree with three children into a multistage optimization graph. The ordered pair $(T^{\text{SIMD}}_{\text{mixed-4}}, T^{\text{SPMD}}_{\text{mixed-4}})$ is obtained from the solution of the multistage optimization problem shown in Fig. 8b.

A node with m children is represented by a multistage optimization problem with $2m + 1$ stages: m stages to represent the nodes, m stages to represent possible mode switches, and one stage to represent the final states at the end of the multistage optimization graph. Thus, $2m + 1 - 2 \cong 2m$ iterations of Moore's algorithm are required to find a solution. For each iteration corresponding to a reduction of the solved portion of the graph with a possible mode switch, $2^2 = 4$ comparisons and $2^2 = 4$ assignments are needed, while for each iteration corresponding to a reduction of the solved portion of the graph with a node execution, no comparisons and $2^2 = 4$ assignments are needed. Thus, finding the minimum mixed-algorithm execution time by this approach has a sequential time complexity of $8m + 4m = 12m = O(m)$ time.

By recording the minimum paths selected at each iteration, the mode of parallelism for each stage is selected. Each edge in the minimum-cost path selected corresponds to performing a node in SIMD, performing a node in SPMD, performing a mode switch, or staying in the same mode. When the multistage optimization is completed for all the children of a node in the SIMD/SPMD trade-off tree, the execution time information is used to generate the execution time for the parent node. The mul-

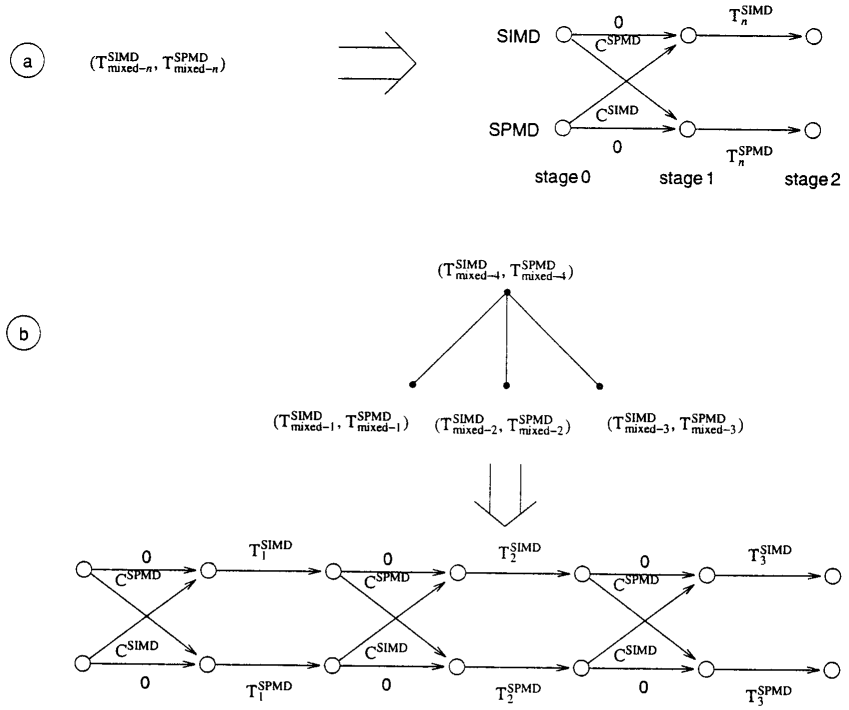

FIG. 8. Transformation from flow-analysis tree to multistage graph. (a) Transforming a node into two stages. (b) Transforming a node with three children into a multistage graph.

tistage approach is then applied to the parent node and its siblings. In this way, the analysis works up the trade-off tree until execution costs are obtained for the root node.

5.4. An Example of Optimal Mixed-Algorithm Selection

As an example of an optimal assignment of modes to the segments of a data-parallel program, consider the SIMD/SPMD trade-off tree in Fig. 9, composed of a `for` loop, where one of the nodes of the loop is an `if` construct (the first child of the root node is the `for_init` for the `for` loop). For this example, let $Q = 10$, $C^{SIMD} = 4$, and $C^{SPMD} = 2$. For the `if` construct, assuming that the outcome of the conditional tests are not mutually independent, let $p_{then} = 0.1$, $P_{then} = 0.1$, and $P_{else} = 0.8$.

In Fig. 9a, the SIMD/SPMD trade-off tree for the program is shown, where the ordered pairs for the leaf nodes have been determined. After the first transformation, the leaf nodes for the `then` clause have been combined to a single node, illustrated in Fig. 9b. Because the `then` node is part of a data-conditional construct, the ordered pair $(T^{SIMD}_{mixed\text{-}then}, T^{SPMD}_{mixed\text{-}then})$ is simply the ordered sum of its children, $(8 + 20 + 2, 10 + 17 + 3) = (30, 30)$. Similarly, for the `else` clause, the ordered pair $(T^{SIMD}_{mixed\text{-}else}, T^{SPMD}_{mixed\text{-}else})$ is $(5 + 5, 18 + 12) = (10, 30)$.

In Fig. 9c, an ordered pair for the `if` construct can then be found, as indicated in step 2. Specifically,

$$T^{SIMD}_{mixed\text{-}if} = (1 - P_{else})T^{SIMD}_{mixed\text{-}then} + (1 - P_{then})T^{SIMD}_{mixed\text{-}else} = (0.2 \times 30) + (0.9 \times 10) = 15$$

$$T^{SPMD}_{mixed\text{-}if} = p_{then}T^{SPMD}_{mixed\text{-}then} + (1 - p_{then})T^{SPMD}_{mixed\text{-}else} = (0.1 \times 30) + (0.9 \times 30) = 30.$$

The ordered pair $(T^{SIMD}_{mixed\text{-}for}, T^{SPMD}_{mixed\text{-}for})$ can then be found using the multistage optimization approach and step 3. Using the multistage optimization approach for the children of the `for` construct, it is determined that $T^{SIMD}_{mixed\text{-}iteration}$, the minimum execution time for a single iteration beginning and ending in SIMD, is obtained by implementing the first block in SPMD mode, the `if` construct is SIMD mode, and the last block in SIMD mode. There will need to be a mode switch before the `if` node (C^{SIMD}) and before the first node of the loop body (C^{SPMD}). Thus,

$$T^{SIMD}_{mixed\text{-}iteration} = C^{SPMD}(=2) + (10, 2) + C^{SIMD}(=4) + (15, 30) + (15, 5) = 2 + 2 + 4 + 15 + 15 = 38.$$

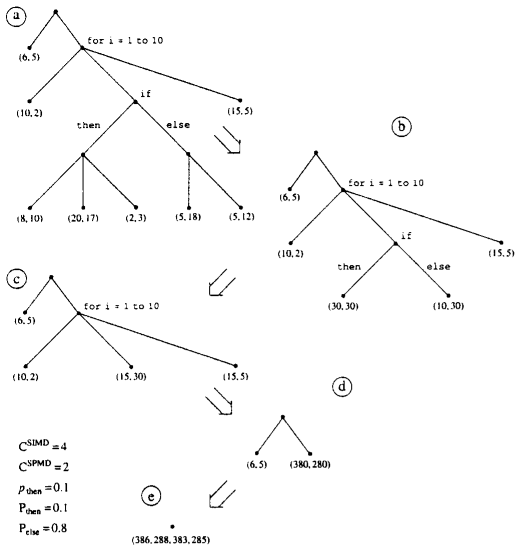

FIG. 9. Mixed-algorithm analysis example for a parallel code segment: (a) time assignments made for leaf blocks; (b) time assignments for `then` and `else` nodes calculated; (c) time assignments for `if` node calculated; (d) time assignments for `for` loop calculated (except for initialization); (e) time assignments for root node calculated.

The multistage optimization approach also determines that for $T^{SPMD}_{mixed\text{-}iteration}$, the first block is implemented in SPMD mode, the `if` construct in SIMD mode, and the last block in SPMD mode:

$$T^{SPMD}_{mixed\text{-}iteration} = (10, 2) + C^{SIMD}(=4) + (15, 30) + C^{SPMD}(=2) + (15, 5)$$
$$= 2 + 4 + 15 + 2 + 5 = 28.$$

Applying step 3 yields:

$$T^{SIMD}_{mixed\text{-}for} = Q \times T^{SIMD}_{mixed\text{-}iteration} = 10 \times 38 = 380$$

$$T^{SPMD}_{mixed\text{-}for} = Q \times T^{SPMD}_{mixed\text{-}iteration} = 10 \times 28 = 280.$$

This is shown in Fig. 9d.

By applying step 4, using the multistage optimization approach, and including mode switch costs the values for the root node are obtained:

$$(T^{SIMD/SIMD}_{root}, T^{SIMD/SPMD}_{root}, T^{SPMD/SIMD}_{root}, T^{SPMD/SPMD}_{root}) = (386, 288, 383, 285).$$

This is shown in Fig. 9e.

To determine the value of $T^{SPMD/SIMD}_{root}$, the first node in Fig. 9d, representing the `for_init`, requires a time of 5 in SIMD. The second node, representing a `for` loop to be executed in SPMD, requires 380. However, the calculation of the value of 380 for the body of the `for` loop includes the cost of switching from SIMD to SPMD needed at the beginning of the loop body. This mode switch can be bypassed, as discussed in Subsection 5.2 and illustrated in Fig. 6, saving 2 time units. Thus, $T^{SPMD/SIMD}_{root} = 5 - 2 + 380 = 383$.

For comparison, the estimated execution time for a pure SIMD or pure SPMD implementation for this example is 406 and 375, respectively.

The mode of parallelism for each portion of the tree can then be chosen, as illustrated in Fig. 10. For this example, the first child and the last child of the `for` node are implemented in SPMD mode. Because the last child of the `for` node is implemented in SPMD, by definition the `for` construct is implemented in SPMD. The `if` node and all its descendents are implemented in SIMD mode.

6. SUMMARY AND FUTURE WORK

The block-based mode selection model proposed in this paper establishes a framework on which to build static source-code analysis techniques for the selection of parallel modes in a mixed-mode context. The model consists of an algorithm for determining information in an SIMD/SPMD trade-off tree, which is then combined using a set of rules and by employing a multistage optimization technique to determine the minimum mixed-algorithm execution time for a sequence of nodes. With this approach, parallel programs written in a mode-independent language can be executed on a mixed-mode machine with each program segment using the most appropriate mode of parallelism for minimum total program execution time.

There are various extensions to the model that form the basis for future work. The set of control and data-conditional constructs can be expanded to include other useful constructs, e.g., `while` statements, `case` statements, and function calls. The probabilistic model intro-

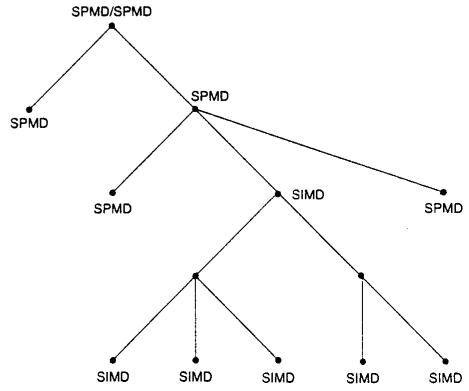

FIG. 10. Selection of parallel modes for the example of Fig. 9.

duced in this paper can be enhanced to consider more fully the effect of juxtaposed blocks on overall execution time. The framework can also include a more complete model of CU/PE overlap for SIMD operation. Other research may involve more practical aspects of the analysis, for example, the details of incorporating the techniques presented here into a parallel compiler. Methods for estimating the parameters used in the model, when they are not deterministic, must be developed.

For a practical implementation in a heterogeneous environment composed of a suite of parallel machines, the time to move data among machines will not be a constant for all blocks, and decisions at one point will impact the quantity future data movements (and thus the amount of time required for intermachine data transfers). Therefore, an important extension to this study is to examine the case where the cost of switching machines/modes is not constant, but depends on the size, location, and usage of data within the program [40].

Another research area that is the subject of future work is the impact on the analysis of including computation models other than SIMD and SPMD, e.g., MIMD and vector processing. The incorporation of other parallel/vector models would form the basis for future efforts to provide programming tools for heterogeneous systems.

ACKNOWLEDGMENTS

The authors thank J. Armstrong, R. Born, T. Casavant, W. Cohen, H. Dietz, A. Maciejewski, W. Nation, R. Palmer, J. M. Siegel, R. Ulrey, R. Wolski, and especially G. Saghi for their valuable comments. A preliminary version of portions of this research was presented at the Workshop on Heterogeneous Processing, April 1993.

REFERENCES

1. Aho, A., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
2. Antonio, J. K., Tsai, W. K., and Huang, G. M. A highly parallel algorithm for multistage optimization problems and shortest path problems. *J. Parallel Distrib. Comput.* **12,** 3 (July 1991), 213–222.
3. Auguin, M., and Boeri, F. The OPSILA computer. In Consard, M. (Ed.). *Parallel Languages and Architectures*. Elsevier, Amsterdam/New York, 1986, pp. 143–153.
4. Auguin, M., and Boeri, F. Experiments on a parallel SIMD/SPMD architecture and its programming. *Proc. France–Japan Artificial Intelligence and Computer Science Symp. '87*. Nov. 1987, pp. 385–411.
5. Berg, T. B., Kim, S. D., and Siegel, H. J. Limitations imposed on mixed-mode performance of optimized phases due to temporal juxtaposition. *J. Parallel Distrib. Comput.* **13,** 2 (Oct. 1991), 154–169.
6. Berg, T. B., and Siegel, H. J. Instruction execution trade-offs for SIMD vs. MIMD vs. mixed-mode parallelism. *Proc. 5th Int'l Parallel Processing Symp.* May 1991, pp. 301–308.
7. Browne, J., Azam, M., and Sobec, S. CODE: A unified approach to parallel programming. *IEEE Software* **6,** 4 (July 1989), 10–17.
8. Browne, J. C., Tripathi, A. R., Fedak, S., Adiga, A., and Kapur, R. N. A language for specification and programming of reconfigurable parallel computation structures. *Proc. 1982 Int'l Conf. Parallel Processing*. Aug. 1982, pp. 142–149.
9. Bronson, E. C., Casavant, T. L., and Jamieson, L. H. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *IEEE Trans. Parallel Distrib. Systems* **1,** 2 (Apr. 1990), 195–205.
10. Bryson, A. E., and Ho, Y.-C. *Applied Optimal Control*. Hempshire, Washington, DC, 1975.
11. Chen, M., and Cowie, J. Prototyping Fortran-90 compilers for massively parallel machines. *Proc. ACM SIGPLAN 1992 Conf. Programming Language Design and Implementation*. June 1992, pp. 94–105.
12. Darema, F., George, D. A., Norton, V. A., and Pfister, G. F. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Comput.* **7,** 1 (Nov. 1988), 430–433.
13. David, H. A. *Order Statistics*, 2nd ed., Wiley, New York, 1981.
14. Dijkstra, E. A note on two problems in connexion with graphs. *Numer. Math.* **1** (1959), 269–271.
15. Dietz, H. G., and Klappholz, D. Refined C: A sequential language for parallel programming. *Proc. 1985 Int'l Conf. Parallel Processing*. Aug. 1985, pp. 442–449.
16. Dietz, H. G., Zaafrani, A., and O'Keefe, M. T. Static scheduling for barrier MIMD architectures. *J. Supercomputing* **5** (1992), 263–289.
17. Fineberg, S. A., Casavant, T. L., and Siegel, H. J. Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting. *J. Parallel Distrib. Comput.* **11,** 3 (Mar. 1991), 239–251.
18. Freund, R. F. Optimal selection theory for superconcurrency. *Proc. Supercomputing '89*. Nov. 1989, pp. 699–703.
19. Freund, R. F. SuperC or distributed heterogeneous HPC. *Comput. Systems Engrg.* **2,** 4 (1991), 349–355.
20. Freund, R. F., and Siegel, H. J. Heterogeneous processing. *Computer*, Special Issue on Heterogeneous Processing. **26,** 6 (June 1993), 13–17.
21. Giolmas, N., Watson, D. W., Chelberg, D. M., and Siegel, H. J. A parallel approach to hybrid range image segmentation. *Proc. 6th Int'l Parallel Processing Symp.* Mar. 1992, pp. 334–342.
22. Gupta, M., and Banerjee, P. Compile-time estimation of communication costs on multicomputers. *Proc. 6th Int'l Parallel Processing Symp.* Mar. 1992, pp. 470–475.
23. High Performance Fortran Forum. Draft: High performance Fortran language specification. *High Performance Fortran Forum*. Sept. 1992.
24. Hillis, W. D. and Steele, G. L., Jr. Data parallel algorithms. *Comm. ACM* **29,** 12 (Dec. 1986), 1170–1183.
25. Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., and Tseng, C.-W. An overview of the Fortran D programming system. *Preliminary Proc. 4th Workshop on Languages and Compilers for Parallel Computing*. Aug. 1991, pp. e1–e17.
26. Jamieson, L. H. Characterizing parallel algorithms. In Jamieson, L. H., Gannon, D. B., and Douglass, R. J. (Eds.). *The Characteristics of Parallel Algorithms*. MIT Press, Cambridge, MA, 1987, pp. 65–100.
27. Khokhar, A. A., Prasanna, V. K., Shaaban, M. E., and Wang, C.-L. Heterogeneous computing: challenges and opportunities. *Computer*, Special Issue on Heterogeneous Processing. **26,** 6 (June 1993), 18–27.
28. Kim, S. D., Nichols, M. A., and Siegel, H. J. Modeling overlapped operation between the control unit and processing elements in an SIMD machine. *J. Parallel Distrib. Comput.*, Special Issue on Modeling of Parallel Computers. **12,** 4 (Aug. 1991), 329–342.
29. Midkiff, S. P., and Padua, D. A. Issues in the optimization of parallel programs. *Proc. 1990 Int'l Conf. Parallel Processing*. **2,** Aug. 1990, pp. 105–113.
30. Moore, E. F. The shortest paths through a maze. *Proc. Int'l Symp. Theory of Switching*. 1957, pp. 285–292.

31. Nichols, M. A., Siegel, H. J., and Dietz, H. G. Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. *IEEE Trans. Parallel Distrib. Systems.* **4,** 2 (Feb. 1993), 222–234.

32. Papoulis, A. *Probability, Random Variables, and Stochastic Processes.* McGraw–Hill, New York, 1984.

33. Philippsen, M., Warschko, T., Tichy, W. F., and Herter, C. Project Triton: Towards improved programmability of parallel machines. *Proc. 26th Hawaii Int'l Conf. System Sciences.* Jan. 1993, pp. 192–201.

34. Qin, B., Sholl, H. A., and Ammar, R. A. Micro time cost analysis of parallel computations. *IEEE Trans. Comput.* **40,** 5 (May 1991), 613–628.

35. Siegel, H. J., Armstrong, J. B., and Watson, D. W. Mapping computer vision related tasks onto reconfigurable parallel processing systems. *Computer* **25,** 2 (Feb. 1992), 54–63.

36. Siegel, H. J., Schwederski, T., Nation, W. G., Armstrong, J. B., Wang, L., Kuehn, J. T., Gupta, R., Allemang, M. D., Meyer, D. G., and Watson, D. W. The design and prototyping of the PASM reconfigurable parallel processing system. In Zomaya, A. (Ed.). *Parallel Computing: Paradigms and Applications,* Chapman and Hall, London, UK, 1994, to appear.

37. Sunderam, V. S. Design issues in heterogeneous network computing, revised edition of proceedings. *Proc. Workshop on Heterogeneous Processing (WHP '92).* May 1992, pp. 101–112.

38. Tucker, L. W., and Robertson, G. G. Architecture and applications of the Connection Machine. *Computer* **21,** 8 (Aug. 1988), 26–38.

39. Wang, M. C., Kim, S. D., Nichols, M. A., Freund, R. F., Siegel, H. J., and Nation, W. G. Augmenting the optimal selection theory for superconcurrency. *Proc. Workshop on Heterogeneous Processing (WHP '92).* May 1992, pp. 13–21.

40. Watson, D. W., Antonio, J. K., Siegel, H. J., and Atallah, M. J. Static program decomposition among machines in an SIMD/SPMD heterogeneous environment with non-constant mode switching costs. *Proc. Workshop on Heterogeneous Processing (WHP '94).* Apr. 1994, pp. 58–65.

41. Zima, H. P., Bast, H.-J., and Gerndt, M. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Comput.* **6,** 1 (Jan. 1988), 1–18.

DANIEL W. WATSON is an assistant professor in the Department of Computer Science at Utah State University. He received a BSEE degree from Tennessee Technological University in 1985, and the MSEE and Ph.D. degrees from Purdue University in 1989 and 1993, respectively. He has co-authored several publications on parallel processing, and his current research interests include heterogeneous computing environments and automatic parallel mode selection techniques. He is a member of the IEEE and IEEE Computer Society, and of Gamma Beta Phi, Tau Beta Pi, and Eta Kappa Nu honorary societies.

HOWARD J. SIEGEL is a professor and Coordinator of the Parallel Processing Laboratory in the School of Electrical Engineering at Purdue University. He received two BS degrees from MIT (1972), and the MA (1974), MSE (1974), and Ph.D. (1977) degrees from Princeton. He has co-authored over 190 technical papers and authored one book. His current research interests include heterogeneous computing, the use and design of the PASM reconfigurable mixed-mode parallel machine, and interconnection networks. He is a Fellow of the IEEE and was Co-editor-in-Chief of the *Journal of Parallel and Distributed Computing* (1989–1991). He is currently on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers.*

JOHN K. ANTONIO is an assistant professor in the School of Electrical Engineering at Purdue University. He received the BS degree (with honors), the MS degree, and the Ph.D., all in Electrical Engineering, from Texas A&M University, College Station in 1984, 1986, and 1989, respectively. His current research interests include heterogeneous computing, analysis and design of parallel algorithms, and computational aspects of control and optimization. He is a member of the IEEE computer society and the IEEE control systems society. Also, he is a member of the Tau Beta Pi, Eta Kappa Nu, and Phi Kappa Phi honorary societies.

MARK A. NICHOLS received the BS 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 MSEE degree from the Georgia Institute of Technology and in 1991 completed the Ph.D. degree in electrical engineering at Purdue University. He is currently with NCR, San Diego, California, where he examines architectural issues regarding parallel processing systems targeted for data-intensive commercial applications such as parallel relational database management. His research interests include parallel architecture modeling, parallel language/compiler design, and interconnection networks. Dr. Nichols is a member of the IEEE Computer Society and the Association for Computing Machinery.

MIKHAIL J. ATALLAH received the BE in Electrical Engineering from the American University, Beirut, Lebanon, in 1975 and the MSE and Ph.D. in Electrical Engineering and Computer Science from Johns Hopkins University, Baltimore, Maryland, in 1980 and 1982, respectively. In August 1982 he joined Purdue University, West Lafayette, Indiana, where he is currently professor of Computer Science. In 1985 he received a Presidential Young Investigator award from the National Science Foundation. His research interests include the design and analysis of algorithms, parallel computation, and computational geometry. Dr. Atallah is a member of the Association for Computing Machinery, the Society for Industrial and Applied Mathematics, and a senior member of the Institute of Electrical and Electronics Engineers. He serves on the editorial boards of the journals *Computational Geometry: Theory and Applications, Information Processing Letters, International Journal on Computational Geometry and Applications, Journal of Parallel and Distributed Computing, Methods of Logic in Computer Science, Parallel Processing Letters, SIAM Journal on Computing,* and was guest editor for a special issue of Algorithmica on "Computational Geometry." He has also served on many conference program committees, and state and federal panels.