Compiler Techniques for Increasing CU/PE Overlap in SIMD Machines

Gene Saghi

Microelectronics Research Center
Dept. of Electrical Engineering
University of Idaho
Moscow, ID 83844-1023, USA

Howard Jay Siegel

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285, USA

Abstract

Conceptually, an SIMD machine has the capability to overlap operation of the control unit (CU) with the operation of the processing elements (PEs). Computation of a single program is most efficient when the work load of the CU and the PEs is balanced. Load balancing between the CU and PEs is accomplished by migrating certain computations (e.g., PE-common array index calculations, loop index variable manipulation) from the PEs to the CU and vice versa. The goal of this research is to develop some of the techniques needed for the automatic specification of CU/PE overlap at compile time. As a proof of concept, the ELP compiler has been modified to support experimentation with CU/PE overlap.

1 Introduction

SIMD machines continue to provide a flexible, cost-effective means of solving real applications in many problem domains (e.g., see [6, 7]) and are therefore likely to remain one popular approach to parallel processing. Conceptually, an SIMD machine has the capability that the operation of the CU(control unit) can be overlapped with the operation of the PEs (processor/memory pairs); e.g., Illiac IV [2], MPP [4]. If the CU is designed to have an independent functional unit for the broadcasting of instructions to the PEs, CU processing can proceed independently of this broadcasting. This independence provides the capability of overlapped operation of the CU and the PEs. An SIMD machine with this capability is called an overlapped SIMD (OSIMD) machine [8].

This research was supported by NRaD under contract number N68786-91-D-1799 and by Rome Laboratory under contract number F30602-92-C-0150. The equipment used was supported in part by the National Science Foundation under grant number CDA-9015696.

In [8], a mathematical model of overlapped operation between the CU and PEs in an OSIMD machine is presented. It has been shown that manual analysis of SIMD programs can be employed to decrease the program execution time by increasing the CU/PE overlap (e.g., [12]). The research presented here augments and builds upon this earlier work. The characteristics of OSIMD machines and programs are analyzed to determine the instructions that can be migrated between the CU and the PEs. In particular, the interaction between CU and PEs is studied to determine when synchronization is necessary. Using this information, it is shown that the model of [8] cannot be applied to programs that contain data-dependent control flow constructs. A practical heuristic is introduced that can be used to increase CU/PE overlap in the general case. Furthermore, a new hybrid method is presented that synergistically applies the mathematical model of [8] and the heuristic introduced here to the problem of increasing CU/PE overlap in OSIMD machines.

The goal of this research is to design some of the techniques needed for the automatic specification of CU/PE overlap at compile time. As proof of concept, the ELP (Explicit Language for Parallelism) compiler [11], a native-code compiler developed at Purdue, has been modified to generate code for experimentation with CU/PE overlap on the PASM [14] parallel processing system prototype. Other machines capable of CU/PE overlapped SIMD operation can achieve similar benefits using the concepts presented here.

Section 2 introduces CU/PE overlapped operation in SIMD machines, the CU architectural model, and the structure of the CU and PE instruction streams. The need for synchronization between the CU and the PEs is discussed in Section 3. A compiler can use the results of such an analysis to determine the CU and PE instruction streams. Section 4 considers the specification of CU/PE overlap by compiler. Increasing CU/PE

overlap on PASM using a modified ELP compiler is discussed in Section 5.

2 System architectural model

A detailed model of the interaction between CU and PEs in an OSIMD machine is summarized from [8] in this section. The model assumes a physically distributed memory configuration (e.g., MasPar MP-1 and MP-2 [5, 10], Thinking Machines CM2 [15]), but the research results of the subsequent sections are equally applicable to a physically global memory configuration (e.g., STARAN [3], TRAC [9]).

The CU specifies a set of instructions to be broadcast to the PEs as an SIMD code block. An SIMD code block consists of any nonzero number of instructions and an SIMD program may consist of many SIMD code blocks. The CU initiates parallel execution of instructions on the PEs by sending SIMD code blocks from its fetch unit (FU) memory to its FU FIFO queue (see Fig. 1). Once in the FU FIFO, the instructions are broadcast to the PEs without any need for further action by the CU. While the PEs are dequeuing and executing instructions from the FU FIFO, the CU processor (CU CPU) is free to perform computations of its own, allowing CU/PE overlap.

An example of one way in which the CU of an SIMD machine can be structured to facilitate CU/PE overlap is shown in Fig. 1. The CU contains its own memory from which the CU CPU reads CU code blocks (blocks executed by the CU CPU) consisting of control flow and scalar computation instructions and associated data. The FU contains a separate memory used to store the SIMD code blocks. The CU coordinates the broadcast of SIMD code blocks to the PEs by means of its fetch unit controller (FC). The FC, at the direction of the CU CPU, fetches SIMD code blocks from the FU memory and loads them into the FU FIFO. Thus, the dequeuing of instructions by the PEs can be overlapped with the execution of instructions by the CU CPU.

In addition to directing the FC to load SIMD code blocks, the CU CPU also has the ability to directly load the FU FIFO. This option allows greater flexibility in the division of work between the CU and PEs by allowing the CU to broadcast the results of its computations to the PEs. For example, if the PEs all need the result of the same address offset computation, the CU could perform this computation and broadcast the result to the PEs. The CU is only allowed to directly load the FU FIFO when the FC is idle (i.e., not loading a SIMD code block into the FU FIFO).

Each CU code block can begin with instructions for initiating the broadcast of an SIMD code block. These FC control instructions typically make up a small section of a code block (e.g., for PASM only two instructions).

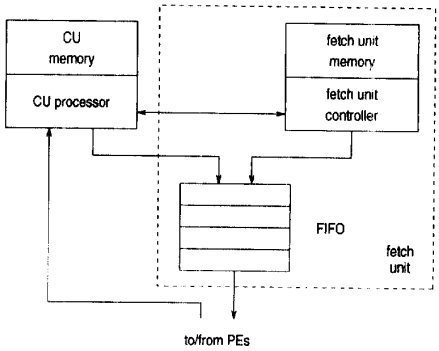

Fig. 1: Simplified diagram of CU data and control flow.

Assume that the CU CPU instruction stream is composed of the sequence of CU code blocks C_0, C_1, C_2, $\cdots$, C_i, $\cdots$ and that the PE instruction stream is composed of the sequence of SIMD code blocks S_1, S_2, $\cdots$, S_i, $\cdots$. The rule for the formation of code blocks is that the execution of C_i on the CU CPU can be overlapped with S_i on the PEs, and the execution of C_{i-1} precedes the execution of S_i. Thus, there are no dependencies between S_i and C_i and these two code blocks are referred to collectively as a combined code block.

Let CU idle time be the time that the CU CPU is idle and cannot proceed to its next instruction because the FC is still loading the last SIMD code block into the FU FIFO or because the CU must synchronize with the PEs. PE idle time is defined as the time that the PEs are idle because the FIFO is empty and there are no instructions for the PEs to dequeue.

3 Determining the instruction streams

The ability of the FC to load SIMD code blocks into the FU FIFO coupled with the ability of the PEs to dequeue instructions from the FU FIFO autonomously allows the CU CPU and the FU to operate asynchronously with respect to one another. However, there are times when the CU CPU and the PEs must synchronize. For example, if the conditional expression for a `while` loop construct depends on a variable local to each PE, the CU CPU must determine if the condition is true on *any* PE before it knows whether to instruct the FU to load the instructions that make up the body of the loop (or to flush the FU FIFO if the loop body instructions were already loaded and the condition was false for all PEs). Places in an SIMD program where such synchronization is necessary are called CU-CPU/PE synchronization points. Combined code blocks cannot span CU-CPU/PE synchro-

nization points because that would create dependencies between S_i and C_i. Thus, a compiler must be able to determine these points before it can derive the CU and PE instruction streams. Furthermore, the code in a SIMD program that lies between two synchronization points can be transformed into at most one combined code block. A detailed analysis of the impact of CU-CPU/PE synchronization points on the overlapped operation of SIMD machines can be found in [13].

4 Increasing CU/PE overlap

4.1 Load balancing

Maximizing CU/PE overlap is accomplished by minimizing CU and PE idle times. By migrating computations from the PEs to the CU or vice versa, the CU and PE idle times can be reduced or eliminated. For example, if PE idle time exists for a combined code block, then migrating computations from the CU to the PEs will reduce the amount of PE idle time. The question of which instructions can be migrated is now addressed.

In an SIMD machine, variables can be classified as scalar-valued or vector-valued. At any given point in time, a scalar-valued variable is allowed to have only one value. Such a variable is called a mono (as in mono-valued) [11]. A variable that is allowed to simultaneously have different values on different PEs is called a poly (as in poly-valued) [11]. Poly data is stored in the PE memories and mono data is typically stored in the CU memory. Computations involving only constants and monos are performed exclusively on the CU, while computations involving only constants and polys are performed on the PEs. For computations involving both monos and polys, the CU performs the mono computation and broadcasts the result to the PEs, where the remainder of the mixed computation is performed. To allow the compiler more flexibility in the migration of instructions, a different approach is used here. As before, polys are stored only on the PEs. However, monos are stored both on the CU and on the PEs.

The compiler maintains a data-location table for all mono variables. The data-location table indicates for each mono variable the locations where the current value of the variable exists. One of three entries is possible for every mono variable: "CU," "PE," or "both." When an assignment to a mono variable is performed on the CU, the table entry is set to "CU." Likewise, when an assignment to a mono variable is performed on the PEs, the table entry is set to "PE" (a given mono has the same value on all PEs). When an update of the mono variable is made from the current location to the other location, the data-location table entry is set to "both."

Assume that all assignments to monos are initially performed on the CU. Mono values on the PEs are not necessarily kept current at all times. If some computations are migrated to the PEs, the compiler is responsible for making sure that the PEs have the current values of any mono variables involved. Thus, the compiler may have to insert code into the CU and PE instruction streams to update PE mono values where appropriate. Likewise, if a mono variable was last modified on the PEs and the CU is to perform a computation involving that variable, the compiler will have to insert code into the CU and PE instruction streams to cause the CU copy of that variable to be updated. The impact of adding instructions to the instruction streams must be included in load balancing decisions made by the compiler.

By allowing for storage of monos on both the CU and the PEs, the compiler has more freedom to schedule computations on either. Thus, instructions that can be migrated include those involved in the evaluation of any expression or subexpression that does not involve poly variables.

4.2 Applying CU/PE overlap analysis

In [8], a mathematical model is presented that can be used to determine the amount of CU idle time and PE idle time in a single combined code block and in a sequence of combined code blocks. There, the simplifying assumption is made that the exact sequence of combined code blocks is known (at compile time, if the compiler is to maximize the CU/PE overlap). In general, the execution path through a program is known only at execution time because there may exist control constructs dependent on input data. This subsection investigates the viability of increasing CU/PE overlap in this general case.

Consider the following code segment where `a`, `b`, and `i` are mono variables and `p` is a poly variable:

```
a[0] = 0;
a[1] = 1;
for (i=2; i<b; i++)
    a[i] = a[i-1] + a[i-2];
p = 0;
```

For this example, the first CU code block would contain the entire loop, while the first SIMD code block would contain the single poly assignment statement. Assuming the value of `b` is determined at execution time, a compiler has no way of knowing the the execution time for the CU code block containing the loop. Thus, the CU/PE overlap model in [8] cannot be applied.

One way to address the problem of CU code blocks with unknown execution time is to subdivide each CU

code block into one or more basic CU code blocks, where a basic CU code block is a sequence of CU instructions that can be entered only at the beginning and exited only at the end. Only the first basic CU code block, in a given initial CU code block, has an associated SIMD code block. In this way, the execution time for each basic CU code block can be determined by the compiler. Consider the case of the preceding example. There would be four basic CU code blocks for this case: (1) the first two assignment statements and the loop initialization code, (2) the code to evaluate the conditional and the conditional branch around the loop body, (3) the code for the body of the loop and the branch back to the top of the loop, (4) the instructions to load the SIMD code block (`p = 0`). Although the duration of each basic CU code block can be determined, it is still not possible to determine the sequence of basic CU code blocks that will occur at execution time.

A basic CU code block is required for every SIMD code block (minimally containing FU instructions to load the SIMD code block). However, as discussed above, not every basic CU code block will have a corresponding SIMD code block (of non-zero length). Furthermore, a compiler can only consider one combined code block at a time when trying to increase CU/PE overlap because the execution sequence of basic CU code blocks is not known at compile time. The question then becomes: can a compiler use the information available to improve program performance by increasing CU/PE overlap?

Without loss of generality, assume that the compiler initially assigns as much work to the CU as possible. That is, all computations involving constants and scalar-valued mono variables are assigned to the CU. Now, consider a basic CU code block and an associated SIMD code block. Only one of three conditions can exist when this combined code block is considered in isolation: (1) CU idle time, (2) PE idle time, and (3) no idle time.

First, consider CU idle time. Because the compiler initially assigns as much work as possible to the CU, there is no action that the compiler can take to increase the work load of the CU in this situation. Furthermore, if the compiler moves work from the CU to the PEs, the CU idle time will increase. Thus, the situation cannot be improved and the compiler should take no action.

Now, consider PE idle time. In this case, the compiler can migrate instructions from the CU to the PEs. If too much work is moved, CU idle time can occur. If the compiler moves an amount of work from the CU to the PEs such that the time for the CU to execute its code is equal to the time for the FU to load the SIMD code block into the FIFO, no CU idle time occurs. However, if that same combined code block is executed, for example, 1000 times, the FIFO will fill up (it is assumed that the FU can fill the FIFO faster than the PEs can empty it) and CU idle time will occur. Thus, it is reasonable to have the compiler attempt to migrate code between a basic CU code block and the associated SIMD code block (if one exists) such that the execution times of the two code blocks is as equal as possible.

The no-idle-time case is similar to the case when PE idle time occurs. No idle time implies that the FC is idle, the FU FIFO is not empty when the CU CPU has finished executing a basic CU code block, and no CU-CPU/PE synchronization is required. The CU is free to begin executing instructions from the next CU code block. Thus, the no-idle-time case does not imply that the execution time for a basic CU code block (on the CU CPU) is equal to the execution time for the associated SIMD code block (on the PEs). If the PEs have more work than the CU, repeated execution of the same combined code block or similar combined code blocks could eventually fill the FIFO and cause CU idle time. Thus, once again, a good compromise appears to be balancing the CU and PE work loads as much as possible on one combined code block at a time.

The proposed heuristic is now presented. The compiler determines the instruction streams using four rules: (1) CU-CPU/PE synchronization points determine the initial division of the CU and PE instruction streams into code blocks (i.e., the code between any two consecutive synchronization points forms one combined code block), (2) the initial CU code blocks are further subdivided into basic CU code blocks, (3) the CU is assigned as much work as possible, and (4) the compiler tries to migrate instructions in isolated combined code blocks from the basic CU code block to the associated SIMD code block in such a way as to make the execution times for the two as equal as possible. Any expression or subexpression that does not involve poly variables can be migrated. The compiler may have to insert code to update any mono variables that are involved in the evaluation of a subexpression and that are not current. This overhead must be considered when applying rule 4 above.

Fig. 2a depicts the processing time for a code segment assuming the CU is assigned as much work as possible (there is no corresponding SIMD code block for the second basic CU code block in this example). It can be seen that there exists PE idle time in this example. In Fig. 2b, computations have been migrated from the CU to the PEs to equalize the execution times on the CU and PEs for the first and third combined code

blocks, as per the heuristic described above. The overall execution time has been decreased in this case.

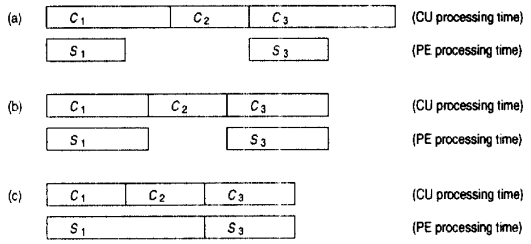

Fig. 2: Execution time for a code segment where (a) the CU work load is maximized, (b) the CU and PE work loads are balanced on a combined code block basis, and (c) the CU and PE work loads are balanced for the sequence of combined blocks.

It can be seen from Fig. 2c that the execution time could be reduced even further for the sequence of combined code blocks shown if more computations could be migrated from the CU to the PEs for the first combined code block. Thus, the heuristic does not always minimize the overall execution time. However, consider a new sequence of the same combined code blocks where the first basic CU code block C_1 (and the corresponding SIMD code block S_1) is executed repeatedly, as it could be in a loop, and C_2 and C_3 are executed only once. For this sequence, the repeated execution of the first combined code block shown in Fig. 2b would result in a smaller overall execution than that of the first combined code block shown in Fig. 2c. This is because in the latter case only the last execution of S_1 would be overlapped with the execution of C_2 and CU idle time would occur during the other iterations. In general, the unpredicatable order of execution of basic CU code blocks does not allow balancing the CU CPU and PE work loads across multiple combined code blocks with consistent results. Therefore, balancing between combined code blocks is not used as part of this heuristic.

It is also possible to combine the heuristic introduced in this section with the mathematical model developed in [8] to form a new hybrid method. The hybrid method consists of adding a rule 3.5 to the heuristic rules. For the hybrid method, the compiler begins by employing the first three rules of the heuristic. Then, it follows rule 3.5, which consists of using the mathematical model to determine how much code should be migrated (if possible) from the CU to the PEs to balance the computation within combined code blocks and across combined code blocks. However, when the compiler encounters a data-dependent branch, or the target of such a branch, it switches to rule 4 because the exact execution sequence of combined code blocks is no longer known and the mathematical model of [8] is no longer applicable. When the compiler reaches a CU-CPU/PE synchronization point, it switches back to rule 3.5 because it knows that the CU CPU and PEs will be synchronized and that the FIFO will therefore be empty at that point. The degree to which the hybrid method can be used to increase CU/PE overlap will depend on the program being compiled (i.e., the existence of instructions that can be migrated).

5 Increasing CU/PE overlap on PASM

5.1 Introduction to ELP and PASM

ELP [11] is an explicitly parallel language designed for programming parallel machines where the user explicitly indicates the parallelism to be employed. The first target machine for ELP is the PASM (partitionable SIMD/MIMD) [14] parallel processing system. A small-scale 30-processor prototype of PASM (with 16 PEs in the computational engine) has been designed and constructed at Purdue University. The PASM design should support at least 1024 PEs. Although the discussion in this section is focused on ELP for PASM, the analyses are applicable to other languages and other OSIMD machines.

An ELP application program is able to perform computations that use the SIMD and MIMD parallelism modes in an interleaved fashion. Here, application programs employing only SIMD parallelism are considered. The ELP compiler has been modified to allow direct experimentation with CU/PE overlap on PASM.

The ELP compiler generates three assembly code output files: one for execution by the CU, one consisting of SIMD code blocks to be broadcast to the PEs by the FU, and one for execution by the PEs (in MIMD mode). Only the CU and FU files are of interest for this study, because these are the only files needed for SIMD programs.

5.2 Code migration potential on PASM

Regardless of whether the CU or the PEs perform a computation, any mono variable values to be used that are not current must be updated first. If the PE copy of a mono variable must be updated, the CU must send its value to the PEs. If the CU copy of a mono variable must be updated, one PE must send its value to the CU (ELP insures that every copy of a mono value on the PEs is the same, as discussed in [11] and [13]).

For the analysis that follows, an ELP subexpression that involves no mono variables is classified as poly. Similarly, an ELP subexpression that involves no poly variables is classified as mono. An ELP subexpression

that involves both mono and poly variables is classified as mixed.

First, consider the case of evaluating a poly subexpression. Because poly variables may be different on every PE, a poly subexpression must be evaluated entirely on the PEs. No code migration is possible.

Now consider the evaluation of a mixed subexpression. Because poly values are involved, some part of the computation must take place on the PEs. If all of the values for the mono variables that appear in the subexpression are current on the PEs, the PEs can perform the entire computation without any communication with the CU CPU. If one or more mono variable values are not current on the PEs, the CU must broadcast at least one value to the PEs via the FU FIFO (this value can be a function of one or more mono variables). In either case, the relevant mono-variable-based calculations are migratable from the CU to the PEs.

Finally, consider the case of evaluating a mono subexpression. If all of the mono variable values involved are current on the PEs, the entire computation can be performed on the PEs. Furthermore, there is no communication required between the CU CPU and the PEs assuming that the CU CPU does not need the result of the computation. If one or more mono variable values are not current on the PEs, communication from the CU CPU to the PEs (via the FU FIFO) is required. As with the mixed-subexpression case, code migration from the CU to the PEs is possible.

Given that code migration is possible for the mixed and mono subexpression cases, a compiler must determine how much (if any) of the code should be migrated to maximize the CU/PE overlap. As discussed earlier, code should only be migrated to the PEs if PE idle time exists. Assuming that PE idle time exists, two factors that must be considered are the locations (i.e., CU versus PE) of valid data for mono variables and whether the result of the subexpression is required by the CU, the PEs, or both. These factors are important because they can generate overhead that impacts the decision to migrate.

5.3 CU/PE overlap extensions to ELP

To support CU/PE overlap experimentation using the ELP compiler, several modifications were made. ELP already determines the initial instruction streams according to CU-CPU/PE synchronization points and assigns as much work to the CU as possible (i.e., commands to broadcast SIMD code blocks and the evaluation of all expressions involving mono variables). The capability of directly partitioning the CU instruction stream into basic CU code blocks at the assembly language level was added to the ELP compiler. Partitioning into basic blocks is described in [1].

The compiler was also modified to report the total execution time for every CU code block and every SIMD code block. The execution times are reported in machine cycles and are inserted as comment lines in the corresponding output file immediately after the associated code block.

Finally, the ELP compiler was modified to generate code that could be executed on either the CU or the PEs (any expression that does not involve a poly variable). Because the evaluation of an ELP expression generally requires more than one assembly language instruction, all such instructions are labeled with a unique number indicating that they must be considered as a set. Interchangeable, possibly different, sets of instructions in the CU and FU output files are assigned the same number. All such statements are output as comment lines.

For example, if `j` is a mono integer variable, the ELP statement "`j++;`" which increments the value of `j`, is output to both the CU and FU output files as:

```
;1      move.l  #_j, a3
;1      addq.l  #1, (a3)
```

where ";1" comments out and labels the pair of instructions as a set. The execution time for these instructions is included in the execution time for the CU code block, but not for the corresponding SIMD code block, because the modified ELP compiler initially assumes all mono calculations will be performed on the CU. The execution time for these two instructions is output to both the CU and FU files as comment lines immediately after the code block execution times. The number assigned to the set of instructions is also used to label the corresponding reported execution time.

The current version of the modified ELP compiler always assumes that the PEs and the CU have the current value of any required mono variable. The implementation of data-location tables is required to guarantee that ELP does not generate code that uses stale values for mono variables and to support the calculation of overhead that needs to be added to the execution time.

The modifications described above are a first step toward developing an ELP compiler that automatically increases CU/PE overlap. Based on a manual analysis of the execution times output by the modified ELP compiler, the assembly language output files can be modified manually to increase the CU/PE overlap by uncommenting either the CU or the corresponding PE instructions. For a given combined code block, if no PE idle time exists, the CU instructions output by the modified ELP compiler are enabled by manually uncommenting them (recall from above that the compiler

initially assumes such code will be executed by the CU). If PE idle time exists, the execution times for each set of instructions that could be migrated are examined manually and one or more sets are selected for migration. Those sets chosen for migration are uncommented in the SIMD code block, while the remaining sets are uncommented in the CU code block. Future work may include the automation of these manual steps and the inclusion of the time needed to update mono variables as a result of migration.

5.4 CU/PE overlap experimentation

Theoretically, the speedup achieved with a compiler that automatically increases CU/PE overlap will range from 1.0 to 2.0 (1.0 for no speedup, 2.0 when there is a perfect balance between the CU and PEs). The experiments conducted on PASM indicate that speedups in the range from 1.3 to 1.8 are typical. The actual speedup for an application program will depend on the availability of mono computations within the program and the way in which these computations are distributed throughout the program (due to the existence of basic code block boundaries that localize code migration). That is, a program with mono computations spaced evenly throughout the program will have much more to gain from automatic CU/PE overlap maximization than, for example, a program that includes large sections of pure poly computation isolated by basic code block boundaries. More information about the PASM experiments is in [13].

6 Conclusions

The mathematical model presented in [8] for determining CU/PE overlap is not applicable to programs that contain data-dependent control flow constructs, including loops and `if then else` statements. Here, a practical heuristic was presented that can be used to increase CU/PE overlap in the general case. In addition, a hybrid method was presented that synergistically applies the mathematical model of [8] and the heuristic introduced here to the problem of finding CU/PE overlap on OSIMD machines. As a proof of concept, an ELP compiler was modified to output information that a programmer can use to determine how to increase CU/PE overlap. Several experiments were performed on PASM to examine CU/PE overlap using the code resulting from the modified compiler analysis. Future work in this area may include further modifications to the compiler to completely automate the process of increasing CU/PE overlap based on the hybrid method.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[2] G. H. Barnes, R. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV computer," *IEEE Trans. Computers*, Vol. C-17, Aug. 1968, pp. 746–757.

[3] K. E. Batcher, "STARAN series E," *1977 Int'l Conf. Parallel Processing*, Aug. 1977, pp. 140–143.

[4] K. E. Batcher, "Array control unit," in *The Massively Parallel Processor*, J. L. Potter, ed., MIT Press, Cambridge, MA, 1985, pp. 170–190.

[5] T. Blank, "The MasPar MP-1 architecture," *IEEE Compcon*, Feb. 1990, pp. 20–24.

[6] T. Blank and J. R. Nickolls, "A Grimm collection of MIMD Fairy tales," *Frontiers '92: The 4th Symp. on the Frontiers of Massively Parallel Computation*, Oct. 1992, pp. 448–457.

[7] D. J. Hunt, "AMT DAP - a processor array in a workstation environment," *Computer Systems Science Engineering*, Vol. 4, Apr. 1989, pp. 107–114.

[8] S. D. Kim, M. A. Nichols, and H. J. Siegel, "Modeling overlapped operation between the control unit and processing elements in an SIMD machine," *J. Parallel Distrib. Computing*, Vol. 12, Aug. 1991, pp. 329–342.

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

[10] J. R. Nickolls, "The design of the MasPar MP-1: A cost effective massively parallel computer," *IEEE Compcon*, Feb. 1990, pp. 25–28.

[11] M. A. Nichols, H. J. Siegel, and H. G. Dietz, "Data management and control-flow aspects of an SIMD/SPMD parallel language," *IEEE Trans. Parallel Distrib. Systems*, Vol. 4, Feb. 1993, pp. 222–234.

[12] G. Saghi, H. J. Siegel, and J. L. Gray, "Predicting performance and selecting modes of parallelism: a case study using cyclic reduction on three parallel machines," *J. Parallel Distrib. Computing*, Vol. 19, Nov. 1993, pp. 219–233.

[13] G. Saghi and H. J. Siegel, *On Increasing CU/PE Overlap in SIMD Machines by Compiler*, Tech. Rep. in prep., Microelectronics Research Center, U. of Idaho.

[14] H. J. Siegel, T. Schwederski, W. G. Nation, J. B. Armstrong, L. Wang, J. T. Kuehn, R. Gupta, M. D. Allemang, D. G. Meyer, and D. W. Watson, "The design and prototyping of the PASM reconfigurable parallel processing system," in *Parallel Computing: Paradigms and Applications*, A. Y. Zomaya ed., Chapman and Hall, London, U.K., 1995, to appear.

[15] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, Vol. 21, Aug. 1988, pp. 26–38.