A Shape Matching Approach for Scheduling Fine-Grained Parallelism†

Brian Malloy
Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906
malloy@cs.clemson.edu

Rajiv Gupta
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
gupta@cs.pitt.edu

Mary Lou Soffa
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260.
soffa@cs.pitt.edu

Abstract - We present a compilation technique for scheduling parallelism on fine grained asynchronous MIMD systems. The shape scheduling algorithm is introduced that utilizes the flexibility of a MIMD system to exploit parallelism within and across basic blocks. Existing techniques exploit parallelism across basic blocks through speculative execution of instructions and code duplication. Our algorithm overlaps the execution of instructions from different basic blocks through matching the shapes of schedules belonging to these basic blocks. In addition, the shape algorithm can reduce the compilation time by increasing the grain size of schedulable units. Experimental results demonstrate that this technique exploits parallelism effectively and that by increasing the grain size the shape algorithm achieves faster compilation times without any significant reduction in program speedup.

1. Introduction

Recent technology has focused on parallelizing a sequential instruction stream to exploit fine grained parallelism using very long instruction word (VLIW) machines[2]. VLIW machines allow the concurrent execution of multiple operations in each instruction. The operations to be executed in each instruction are statically scheduled by the compiler. The lockstep operation of processing units makes the machine intolerant to run-time delays caused by unpredictable events such as memory bank access conflicts. The delay in the completion of any one of the operations in an instruction delays the completion of the instruction. Thus, while the VLIW architectures perform well on scientific applications, their performance can degrade rapidly when faced with factors that decrease run-time predictability.

In order to address the above drawbacks of VLIW machines a number of tightly coupled fine-grained MIMD architectures have been proposed[3,7]. An asynchronous MIMD system is tolerant of delays caused by unpredictable events since the processors are not required to operate in lockstep. Special synchronization and high speed communication hardware is provided to enable high speed processor interaction. The processors can execute relatively independent streams of instructions as well as tightly synchronized instruction streams. Thus, both fine grained and coarse grained parallelism can be exploited by these architectures.

In this paper we develop a compilation technique which exploits the unique features of a fine grained MIMD system. We develop a shape matching algorithm that generates schedules which exploit parallelism across basic blocks. Techniques such as trace scheduling[2] enable the exploitation of fine grained parallelism across basic blocks on VLIW machines. However, they achieve this goal by speculative execution of instructions and code duplication. Speculative execution allows a VLIW machine to achieve greater speedups along likely execution paths at the expense of program paths that are less likely to be executed. Code duplication can lead to code explosion in a trace scheduling compiler. Our shape matching algorithm utilizes the asynchronous nature of the system to exploit parallelism without speculative execution or code duplication. The shape algorithm is based upon the program dependence graph and therefore it is able to move code across control structures in a manner similar to region scheduling[4].

Another cause of concern in compiling for fine grained machines is the cost of the compilation process itself. The shape matching algorithm presented in this paper utilizes the system's ability to exploit both fine grained and coarse grained parallelism to achieve efficiency. We demonstrate that our algorithm functions for various grained scheduling units. Thus, by considering larger grain sizes during scheduling we are able to improve the efficiency of the compilation process.

In a MIMD system the cost of processor synchronization and communication must also be considered during scheduling. We base the shape algorithm on the preferred path selection (PPS) algorithm[6], which attempts to minimize interprocessor communication during the execution of straight line code.

† Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

0-8186-3175-9/92 $3.00 © 1992 IEEE

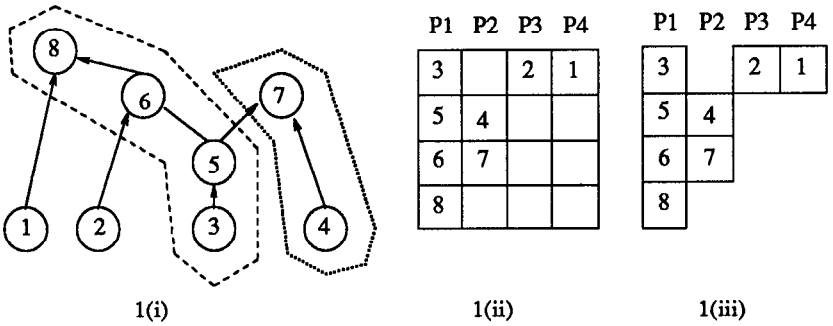

Figure 1. A dag, a possible schedule and a corresponding shape.

2. Overall Approach to Matching Shapes

The program dependence graph[1] (PDG) representation of a program is used by the Shape Algorithm. The PDG expresses control dependences through a control dependence subgraph (CDS) and data dependences through a data dependence subgraph (DDS). There are three kinds of nodes in the CDS: *statement nodes* (S_i), *boolean nodes* (B_i) and region nodes (R_i). A region node points to a set of nodes representing parts of a program that require identical control conditions for execution and the edges connecting regions show the flow of control.

We use the preferred path selection algorithm[6] (PPS) to schedule regions consisting only of statement nodes or straight line code. The PPS algorithm constructs a dag for the nodes in a region, selecting a long path and assigning it to a processor. All nodes along the path are assigned to the same processor, thus enabling the resolution of dependencies among these statements without explicit synchronization. Computation of paths and the assignment of nodes along each path to a processor continues until the dag for this region is scheduled. Since an aspect of this work is to explore the effect of scheduling at various levels of granularity, the nodes in a region may represent (i) operations in the form of intermediate code statements, (ii) statements formed by combining operation nodes, or (iii) basic blocks formed by combining statement nodes.

To illustrate the PPS algorithm, consider the dag and a corresponding schedule in Figure 1 where nodes in the dag represent operations, statements or basic blocks and an edge from node n_i to n_j indicates that the value computed by n_j is required for the computation of n_i. For the schedule shown in Figure 1(ii), the first path (nodes 8, 6, 5 and 3) is assigned to processor 1, the second path (nodes 7 and 4) is assigned to processor 2, the third path (node 2) is assigned to processor 3 and the fourth path (node 1) is assigned to processor 4. Furthermore, without loss of generality, we consider that node 3, assigned to processor 1, begins execution at time 0 and terminates at time 1 and the length of the schedule is 4 time units. In the schedule, node 7 cannot begin execution until node 5 terminates since node 7 requires the value computed at node 5. Thus, node 7 cannot begin execution until time 2 when node 5 terminates. We represent the schedule without insertion of synchronization to guarantee, for example, that node 7 does not begin until node 7 terminates.

As the paths in the dag are scheduled using the PPS, control dependences in the program as well as data dependences between paths form a shape in the schedule. For control dependences, statements that depend on control statements must be scheduled after the control statement. For data dependences, if a definition of a variable x reaches a use of that variable x, then the use must be scheduled after the definition. Figure 1 (iii) illustrates the shape resulting from scheduling the dag using the PPS scheduling algorithm.

Paths are assigned to processors so that they will be scheduled for execution as early as possible, and nodes in a path are placed in contiguous positions in the schedule if data dependences permit. For example in Figure 1, the path containing nodes 7 and 4 cannot begin execution until after node 5 executes.

3. The Shape Matching Algorithm

The shape matching algorithm traverses the PDG creating ordered lists of region nodes RN during its descent until an unscheduled region node is encountered. Procedure *AssignBottom* (Figure 2) is used to create an initial shape in the schedule. Scheduling resumes, progressing from the lowest level of the PDG consisting of unscheduled regions to higher levels containing partially scheduled regions. The shape of a scheduled set of nodes is maintained through variables $Final_i$ and $Initial_i$, the time of the last node to finish execution and the earliest node to begin execution respectively for processor P_i.

When a partially scheduled region is encountered, the unscheduled nodes in this region are partitioned into

sets whose membership is determined by the data dependencies among the scheduled regions and the unscheduled statement nodes. The unscheduled nodes that must execute before scheduled nodes form *TopSet* and unscheduled nodes that must execute after scheduled nodes form the *BottomSet*. The unscheduled nodes that have no data dependencies with the scheduled regions form *DontCareSet*. Nodes in *TopSet* are scheduled by Procedure *AssignTop* so that the formed shape matches the top of the shape of the scheduled nodes; nodes in *BottomSet* are scheduled by *AssignBottom* so that the formed shape matches the bottom of the shape of the scheduled nodes; nodes in *DontCareSet* may be used to match either the top or the bottom of the shape of the unscheduled nodes. Thus, the nodes in the *DontCareSet* are the ones which can be propagated across control structures.

Boolean nodes, encountered in the PDG traversal, represent *For*, *While*, *If-then-else* or *If-then* structures. If a *For* loop is found the data dependences are examined to determine suitability for concurrentization; if suitable, the iterations of the *For* loop are scheduled on the processors. If the loop is not suitable for concurrentization then it is scheduled as if it were a single path in a dag. *While* and *If* statements are scheduled by assigning the statements in their respective regions in the shape matching fashion discussed previously. In scheduling boolean nodes representing *If-Then-Else* structures, the shape matching algorithm determines which of the two regions, the *Then* or *Else* part, to use for the matching since only one of the two will be executed during a given iteration. Profile information is used to determine the more likely path resulting in either the *Then* shape or the *Else* shape becoming part of the scheduled shape. Both regions are scheduled by matching shapes but the less likely path does not impact on future matches. Since the shape of the scheduled regions is maintained in *Initial* and *Final*, these variables are updated only for the region that is more likely to execute.

After scheduling a boolean node, two tasks remain: (i) determine if the structure in the region requires *concurrentization*, and (ii) determine if the shape requires adjustment. The first task is needed when statements within a structure are assigned to different processors. To preserve the semantics of the program, the control conditions and branches of the structure are duplicated so that statements in the concurrentized program have the same conditions for execution that they had in the sequential program. For the second task, the scheduled shape may require adjustment for a concurrentized *While* loop. The two while loops resulting from the concurrentization of a single while loop perform the same number of iterations as the original loop. The shape must be "stretched" to reflect the number of iterations performed so that subsequent assignments can try to balance processor load.

The procedure *AssignBottom*, for assigning nodes in *BottomSet* to processors, is summarized in Figure 2. *AssignBottom* begins by constructing a dag for the nodes in set G and then invokes *ComputeLongPath* to return a long path in the dag. A greedy approach is used to compute the path so that a node is selected primarily on the basis of the level in the dag where the node is located and secondarily on its weight. Nodes at a higher level are chosen over nodes at a lower level to maximize the length of the path. The *weight* of a node is the number of operations included in the node so that "heavier" nodes are preferred to "lighter" ones. Long paths are matched with previously assigned paths in the schedule until all nodes in *G* are scheduled.

```
Procedure AssignBottom(input G :Set of Nodes);
Begin
  Construct a dag for G;
  CurrentLevel:= Level of terminal node at highest level;
  While there are unscheduled nodes ∈ G Do
      ComputeLongPath(L, CurrentLevel); /* Find longest path L */
      Search L, starting at the top, for a use u_i of a variable with a
        scheduled definition that reaches u_i;
      If u_i is Found Then
        Let dist_u be the distance from the top of L to u_i.
        Consult DefUseTable for the exec time t_d of last def d_i
              that reaches u_i
        t_s = t_d - dist_u + 1;
        δ:= 0;  Done:= False;
        While Not Done Do
              i:= 1;
              While Not Done AND i ≤ p Do
                    If ABS(Final[i] - t_s) ≤ δ Then Done:= True;
                    Else i:= i + 1; End If
              End Loop;
              δ:= δ + 1;
        End Loop;
    Else p_i is the processor such that Final[i] is a minimum
    End If
    Assign L to p_i;
    Compute TotalWeight of L, the sum of the weights of the nodes;
    If Final[i] >= t_s Then Final[i]:= Final[i] + TotalWeight
    Else Final[i]:= t_s + TotalWeight; End If
    Update Def/Use table for p_i;
    Possibly set Initial[p_i] to the earliest start time of L;
  End loop;
End AssignBottom;
```

Figure 2. Algorithm to assign nodes in BottomSet.

It is the two innermost *While* loops in *AssignBottom* that actually matches shapes. The goal of shape matching is to choose a processor for an assignment that allows the path being scheduled to start close to its actual start time while creating the smallest hole in the schedule. The actual start time is the time that the first operation in the path begins execution. Due to data and control dependences, the actual start time is always greater

than or equal to the scheduled start time. A *hole* in the schedule occurs when a processor is either idle or is waiting to synchronize. If the long path L that is returned by *ComputeLongPath* does not contain a use of a variable, then *AssignBottom* load balances by assigning L to the processor that has the earliest finish time.

The actions of *AssignTop* in assigning the nodes in TopSet are similar to the action of *AssignBottom* except the long path returned by *ComputeLongPath* is examined, progressing from the bottom of the path to the top, for a definition that reaches a node in the shape formed by scheduled nodes. The actual start time of L is then computed in terms of the start time of the use in the schedule and the position of the definition in L. The path L is then assigned to the processor that will allow L to start close to its actual start time while creating the smallest hole.

Table 1.
Performance of the Shape Matching Algorithm.

Program	Operation		Statement		Basic Block	
	p=8	p=16	p=8	p=16	p=8	p=16
Kernel 1-F	7.99	15.92	7.99	15.92	7.91	15.63
Kernel 1-W	3.79	6.49	5.41	8.84	7.91	15.63
Search	7.61	13.25	7.61	13.25	7.57	13.04
Broadcast	5.46	10.72	5.46	10.72	7.81	15.17
Trap	6.11	8.16	6.98	12.29	6.56	10.03
Vector	3.09	4.12	5.69	7.18	0.99	0.99
MergeSort	4.23	6.61	4.38	6.83	3.11	5.54
Sieve	2.61	2.81	2.61	2.81	2.43	2.79

4. Performance of the Shape Matching Algorithm

The results in Table 1 show that the shape matching algorithm can appreciably improve execution speed of the program. The programs listed in the first 5 rows of Table 1 experienced virtually linear speed up for all three scheduling grains. For example, the Kernel 1-F program experienced a speedup of 7.99 and 15.92 when executed on p=8 and p=16 processors respectively at the operation grain. Both the *Kernel 1-W* and the *Broadcast* programs achieved their best speedup at the basic block grain, while the *Vector* program did not experience good speedup at the basic block grain (0.99 speedup for both 8 and 16 processors) because there were only five basic blocks remaining in the *Vector* program after loop unrolling. Finally, the *Sieve* program did not experience appreciable speedup at any grain because the Sieve of Erosthosthenes algorithm is replete with data dependences so that the gain achieved from the load balancing of the shape matching algorithm is eroded by communications costs, even with a fast communication network.

For most of the benchmark programs, the basic block grained schedules performed as well or better than schedules at the other grains. However, basic block grained schedules can be computed faster than schedules at the other grains. For the benchmark programs, basic block grained schedules were computed from 34 to 146 percent faster than the operation grained schedules.

In further experiments[5], the reported speedups indicate a strong correlation between the simulations and actual executions on the Data General multiprocessor. For example, experiments show that 43,022 cycles are required to simulate the execution of the sequential code for the *Search* program, and 21,537 cycles are required to simulate the execution of the schedule for 2 processors with a speedup of 1.99 over the sequential execution. For the actual execution of the *Search* program on the Data General multiprocessor, an average of 6.18 seconds were required using 1 processor and 3.23 seconds were required for 2 processors producing a speedup of 1.91 over the sequential execution.

References

1. J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and its Use in Optimization," *Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July, 1987.
2. J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, NO. 7, July 1981.
3. Rajiv Gupta, Michael Epstein, and Michael Whelan, "The Design of a RISC based Multiprocessor Chip," *Proceedings of Supercomputing'90, New York*, pp. 920-929, November 1990.
4. Rajiv Gupta and Mary Lou Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 421-431, April, 1990.
5. B. Malloy, "A Fine Grained Approach to Scheduling Asynchronous Multiprocessors," *Technical Report TR92-116*, May 1991.
6. B. Malloy, E.L. Lloyd, and M.L. Soffa., "A Fine Grained Approach to Scheduling Asynchronous Multiprocessors," *4th International Conference on Computing and Information,*, pp. 131-135, May, 1992.
7. A. Wolfe and J.P. Shen , "A Variable Instruction Stream Extension to the VLIW Architecture," *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-14, April 1991.