Modulo Scheduling Aware Cluster Assignment

Eric Stotzer[1] and Ernst L. Leiss[2]

[1] Texas Instruments, 12203 SW Freeway, MS 730, Stafford, Texas, 77477
estotzer@ti.com
[2] Department of Computer Science, University of Houston, Houston, Texas, 77204
coscel@cs.uh.edu

Abstract. Very long instruction word (VLIW) processors are well suited for today's high performance processing applications, which are characterized by mathematically oriented loop kernels and abundant instruction level parallelism. VLIW processors have multiple functional units, which are often partitioned into clusters with local register files. Cluster assignment is the process of partitioning instructions to clusters with the goals of maximizing parallelism and minimizing communication overhead. VLIW compilers use software pipelining to schedule loop body instructions such that multiple loop iterations execute in parallel. A critical phase in a VLIW compiler is cluster assignment for software pipelined loop body instructions. In this paper, we describe modulo scheduling, which is a method for implementing software pipelining. We present a cluster assignment algorithm that is designed to run before modulo scheduling. Finally, we present experimental results that demonstrate the effectiveness of the cluster assignment algorithm for the Texas Instruments TMS320C6x 2-cluster VLIW processor.

Keywords: VLIW, DSP, ILP, Software Pipelining, Modulo Scheduling, Cluster Assignment.

1 Introduction

Signal, video, and image processing applications spend the bulk of their time in compute intensive loop nests that exhibit high degrees of instruction level parallelism (ILP). Very long instruction word (VLIW) processors exploit ILP by executing multiple instructions per cycle. Because they do not have hardware to dynamically find implicit ILP at run-time, VLIW architectures rely on the compiler to statically encode the ILP in the program before its execution. Software pipelining, a powerful loop-based transformation, is key to extracting ILP and exploiting the many functional units on a VLIW.

A VLIW processor has multiple functional units, each capable of executing a RISC-like instruction in parallel. To support two reads and one write per cycle, each functional unit requires two read ports and one write port into a register file. The size of the connection network between the functional units and the register file grows in

proportion to the product of the number of read ports and write ports [7]. Therefore, the connection network between the functional units and the register file can lead to complex routing problems and become cost prohibitive. Hence, VLIW processors usually have split register files [10], where the register set is split into two or more register files, each of which is connected to a set of functional units. A register file along with its associated functional units is called a *cluster*.

On a clustered VLIW processor, instructions are explicitly partitioned to clusters. Functional units in a cluster most efficiently access registers in their associated local register file. To access values across clusters, some form of communication is required, either through dedicated hardware buses, or via explicit copy operations. Communication of a register operand from one cluster to another may use limited resources or cause pipeline stalls. On architectures with homogeneous clusters, the type and number of functional units are the same in each cluster [16]. The goal of *cluster assignment* algorithms is to assign instructions to clusters such that parallelism is maximized and cross-cluster communication is minimized.

For example, the Texas Instruments TMS320C6x (C6x) processor is a VLIW capable of issuing eight instructions in parallel. It has two almost identical clusters (see Figure 1), each with a local register file and four functional units: a multiplier unit (M), a logical/adder unit (L), a shift/logical unit (S), and an address generation/adder unit (D). Each cluster has a dedicated bus that can load or store a 32-bit value to or from memory on each cycle. The clusters have limited ability to access values from each other via cross-path busing.

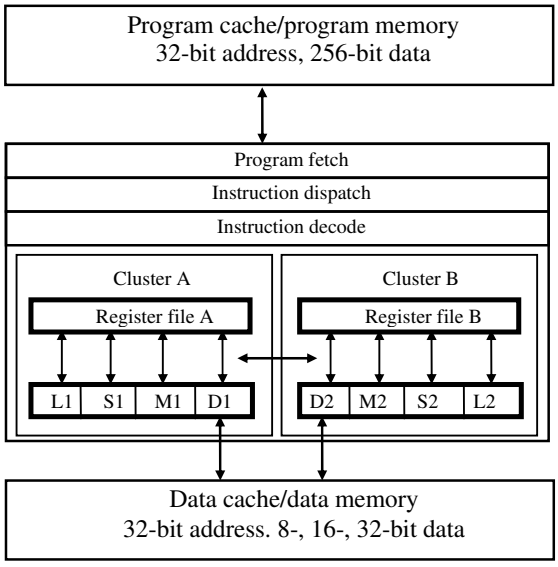

Fig. 1. TMS320C6x Two-Cluster Block Diagram

Modulo scheduling, an algorithm for implementing software pipelining [5], takes the N instructions in a loop and forms an M-stage pipeline as if a vector functional unit were being specially designed to execute the loop body. A modulo-aware cluster assignment algorithm partitions instructions to clusters such that restrictions on modulo scheduling are minimized.

In the following sections we will summarize the related work, provide more background on modulo scheduling and present our cluster assignment algorithm. We will present results that measure how effective our algorithm is at minimizing the overhead of a clustering on modulo scheduling.

2 Synopsis of Related Work

The Multiflow Corporation produced the VLIW Trace computer [4]. The Trace family of computers was available in three sizes where each size replicated a cluster. There were one-, two-, and four-cluster machines. VLIW processors are well suited for high performance scientific and embedded applications [7, 15]. Software pipelining and modulo scheduling are well documented [5, 6, 15, 17, 19].

There are various cluster assignment algorithms; the classical one being the Bottom-Up Greedy algorithm (BUG) [2]. These algorithms map instructions and indirectly registers (or vice versa [2]) to clusters and are performed before instruction scheduling. Other approaches combine cluster assignment, non-pipelined instruction scheduling, and register allocation [9, 12, 13, 14]. These approaches perform cluster assignment without any consideration as to how the instructions will be arranged during modulo scheduling.

We are interested in cluster assignment algorithms for operations within an inner loop that will be software pipelined using modulo scheduling. The cluster assignment algorithm must take into account the modulo scheduling constraints. One approach to this problem is to combine modulo scheduling and cluster assignment at the same time [11]. Another approach is to perform cluster assignment before modulo scheduling, using information about how the loop will pipelined to drive the cluster assignment [8, 13].

3 Software Pipelining

Finding a valid software pipeline is an instruction scheduling problem. A component of the compiler must perform instruction scheduling by examining data dependencies to determine which operations may execute in parallel. Theoretically, instruction scheduling reduces to the NP-complete bounded resource scheduling problem [17]; therefore, heuristics-based scheduling algorithms must be developed. In order to extract parallelism from a program, the relationships between operations must be completely understood. Since only independent operations may execute in parallel, operation dependencies must be determined.

In order to comprehend operation dependencies, a data dependency graph (DDG) is constructed where nodes represent operations and the edges express data

precedence relations between the operations. Cyclic dependence graphs contain back edges caused by loops in the control flow such that results from values computed in one loop iteration are used as inputs in successive iterations. This type of dependence is called a recurrence or loop-carried dependence. The edges in a cyclic dependence graph are augmented with an iteration difference, which is the number of iterations separating the output and input operations of the dependence.

Modulo scheduling is motivated by the development of pipelined hardware functional units. The rate at which new loop iterations are started is called the *Initiation Interval* (II) [5]. The *Minimum Initiation Interval* (MII) is the smallest II for which a valid schedule can be found. The resource bound (ResMII) is determined by the total resource requirements of the operations in the loop. The recurrence bound (RecMII) is determined by loop-carried data dependences. The MII is thus determined as MAX(ResMII, RecMII). Methods for finding the ResMII and the RecMII can be found in [6, 15, 17].

The schedule for a single iteration is divided into a sequence of stages each with a length of II (see Figure 2 where A, B, C, D, and E are the 5 stages of single loop iteration). In the steady state of the execution of the software pipelined loop, each of the stages will execute in parallel. The instruction schedule for a software pipelined loop has three components: a prolog, a kernel, and an epilog. The kernel is the instruction schedule that will execute the steady state. In the kernel, an instruction scheduled at cycle *k* will execute in parallel with all instructions scheduled at cycle *k* modulo II. This is known as the modulo constraint [5] and is the source of the name "modulo scheduling". The prolog and epilog are the instruction schedules that set up and drain the execution of the loop kernel.

Cycle	*Pipeline*					
0	A:1					Prolog
II	B:1	A:2				
II*2	C:1	B:2	A:3			
II*3	D:1	C:2	B:3	A:4		
II*4	E:1	D:2	C:3	B:4	A:5	
			.			Steady State
			.			
			.			
II*(N-5)	E:1	D:2	C:3	B:4	A:5	
II*(N-4)		E:2	D:3	C:4	B:5	
II*(N-3)			E:3	D:4	C:5	
II*(N-2)				E:4	D:5	
II*(N-1)					E:5	Epilog

Fig. 2. Execution trace of a software pipeline

3 Modulo Aware Cluster Assignment

We are concerned with the performance of software pipelined loops on a clustered VLIW processor. Our goal is to develop an algorithm that assigns loop instructions to

clusters in such a way that the constraints placed upon modulo scheduling are minimized. In our approach, ResMII, RecMII,, and thus MII are determined before cluster assignment, assuming a unified non-clustered architecture. MII is passed to the cluster assignment algorithm, which attempts to partition instructions to clusters in a way that does not increase MII. We introduce a new variable called PartMII, which is the MII computed after partitioning instructions to clusters.

We adapted the BUG algorithm, which uses a form of 'greedy' resource scheduling, by making it sensitive to the constraints of modulo scheduling. Starting at the root nodes and passing clustering preferences to predecessors, BUG recursively visits each node in the DDG. If the current node is a leaf or all its predecessors have been scheduled on a cluster, BUG assigns the node to a cluster and inserts the node into a trial schedule. BUG continues this process as it traverses back down the DDG. BUG assigns a functional unit that (greedily) places the node in the trial schedule as early as possible. Since a node represents an instruction, we use the term node to refer interchangeably to a vertex in the DDG or an instruction. In addition, assigning a functional unit to a node assigns that node to the specific cluster that contains the functional unit.

The algorithm is invoked by the function *AssignClusters(DDG, MII)*, which assigns a functional unit to each node in the DDG. MII is used to limit the number of times a resource on a cluster is used. In depth order, the roots of the DDG are passed to the function *Assign(n, succ, U)*, which selects a functional unit for the node *n*. The inputs are a successor node *succ* and a set of function units *U* on which *succ* can execute. The goal is to select the best functional unit for *n* to deliver a result to *succ* executing on one of the functional units in *U*. Assign() is recursively called on each of the predecessors of *n*. A node is assigned a functional unit only after all of its predecessors have been assigned.

```
AssignClusters(graph DDG, int MII) {
  InitModulo(DDG, MII); // Initialize the Modulo Resource Model
  InitSchedule(DDG); // Initialize Linear Schedule Model
  foreach node root in SortDepth(Roots(DDG))
    Assign(root, null, EMPTY);
}
```

(**1**)

```
Assign(node n, node succ, set<unit> U) {
  if (Scheduled(n)) return;
  foreach node pred in SortDepth(Predecessors(n))
    if (notScheduled(pred))
      Assign(pred, n, LikelyUnits(n, succ, U)));
  n.Unit() = BestUnit(n, LikelyUnits(n, succ, U));
  SchedInsert(n, n.Unit(), n.Unit().Cycle());
  AddModulo(n.Unit());
}
```

(**2**)

The function *LikelyUnits(n, succ, U)* considers the current partial trial schedule and returns the likely functional units for node *n*; *n* will deliver its result to a successor node *succ*, where *succ* will ultimately be assigned to a functional unit from the set *U*. The function *FeasibleUnits(n)* returns the set *F* of all feasible functional units on which a node *n* can execute. In our algorithm, we modified this function to consider the current cluster assignment constraints on the MII. We prune from *F* any items that would increase the MII. For example, if MII=3 and a node can execute on the functional unit *x* from cluster *F*, but *x* has already been allocated to three other nodes, then *FeasibleUnits()* will prune *x* from *F*. The function *AddModulo()* keeps track of the partial trial schedules constraints on the MII.

```
set<unit> LikelyUnits(node n, node succ, set<unit> U) {
   cycle_t minCycle = INFINITY;
   set<unit> F = FeasibleUnits(n);
   foreach unit src in F {
      src.Cycle() = INFINITY;
      foreach unit dst in U
         src.Cycle() = min(src.Cycle(), CompletionCycle(n, src, succ, dst));
       minCycle = min(src.Cycle(), minCycle);
   }
   foreach unit src in F
      if (src.Cycle() > minCycle)
         L.remove(src);
   return L;
}
```

(**3**)

The function *StartCycle(n, dst)* computes an estimate of the earliest cycle on which a node *n* can execute in the trial schedule on a functional unit *dst*. It does this by estimating the start cycle, latency, and communication penalty for each predecessor *pred* of *n*. If *pred* is already scheduled, the start cycle is determined by the current trial schedule; otherwise, the start cycle is estimated as the depth of *pred* in the DDG. The communication penalty, calculated by the function *MoveCost(n, src, succ, dst)*, is the number of cycles it takes to communicate a result from node *n* executing on *src* to node *succ* executing on *dst*. When *pred* is unscheduled the communication penalty is estimated as the minimum cost for all functional units in *FeasibleUnits(pred)*. The *Latency(src)* is the number of cycles it takes to complete the execution of node *pred* on the *src* unit. The function *CompletionCycle(n, src, succ, dst)* returns an estimate of the earliest cycle that node *n* can execute on functional unit *src* and deliver its result to the successor node *succ* executing on functional unit *dst*.

```
cycle_t CompletionCycle(node n, unit src, node succ, unit dst) {
      return StartCycle(n, src) + n.Latency(src) +
                                    MoveCost(n, src, succ, dst);
}
```

(**4**)

```
cycle_t StartCycle(node n, unit dst) {
   cycle_t cycle = INFINITY, earlyCycle = 0;
   foreach node pred in Predecessors(n)) {
      if (Scheduled(pred))
        cycle = pred.Cycle() + pred.Latency(pred.Unit()) +
                               MoveCost(pred, pred.Unit(), n, dst);
      else
         foreach unit src in FeasibleUnits(pred))
            cycle = Min(cycle, pred.Depth() + pred.Latency(n, src) +
                                                  MoveCost(pred, src, n, dst));
      earlyCycle = Max(earlyCycle, cycle);
   }
   while (SchedAttempt(n, u, earlyCycle) == false)
      earlyCycle++;
   return earlyCycle;
}
```

(**5**)

4 Experimental Results

In Table 1, we present results for running the cluster assignment algorithm on a set of inner loop kernels extracted from DSP programs. We studied the benefits of the algorithm on a benchmark suite of forty loop kernels. These benchmarks represent the performance critical loop kernels of typical TMS320C6x applications and algorithms: telecom, audio, image processing, and other digital signal processing areas. The loop benchmarks are generally computationally intensive, with large amounts of instruction-level parallelism. Each loop is a DO-loop composed of a single basic block that does not contain a call and does not have an early exit. The trip counters for the loops have been converted to simple downcounters.

We developed a standalone tool that implements our cluster assignment algorithm. The input to this tool is a sequence of instructions that form a loop body. The output of the tool is a software pipelined schedule of the loop and some informative feedback. The structure of the tool is a small front-end, cluster assignment algorithm, and a small back-end that formats the output. The front-end reads an input file, builds a DDG, and classifies each instruction. This information is then passed to the cluster assignment algorithm. The back-end simply formats the output and summarizes the results.

The values in the input file are in a symbolic form. Values are defined only once, and an unlimited number of symbolic values are supported. The simple structure of this tool also facilitates architecture evaluation.

Table 1. Loop kernel benchmark results. MII=MAX(RecMII, ResMII).

Loop Kernel	RecMII	ResMII	MII	PartMII	PartMII-MII
GSM Ck()	8	8	8	10	2
GSM viterbi equalizer viteq()	4	4	4	4	**0**
GSM viterbi ch. dec vitgsm()	2	3	3	3	**0**
GSM viterbi trellis d. vitv32()	3	5	5	5	**0**
VSELP G_codebook()	3	3	3	4	1
VSELP Gl_comp()	0	1	1	1	**0**
VSELP autocorrelation()	9	8	9	9	1
VSELP block_move()	0	2	2	2	**0**
VSELP dot product dotp()	2	2	2	2	**0**
VSELP FIR vfir()	0	10	10	11	1
VSELP high pass filter hpf()	5	5	5	5	**0**
VSELP lag_mac()	0	2	2	2	**0**
VSELP orthoganilization()	1	2	2	2	**0**
VSELP mac vselp_mac()	2	2	2	2	**0**
CELP codebook_search()	4	7	7	7	**0**
CELP IIR filter impulse1()	7	6	7	7	**0**
CELP IIR filter impulse2()	0	10	10	11	1
JPEG dct jpegdct()	0	10	10	13	3
MPEG dct mpegdct()	3	9	9	10	1
Radix 2 FFT radix2()	3	4	4	4	**0**
Radix 4 FTT radix4()	5	9	9	9	**0**
Cyclic redundancy check crc()	6	2	6	6	**0**
Complex FIR cx_fir()	5	5	5	7	2
FIR filter fir()	4	4	4	4	**0**
Gouraud shading gouraud()	0	4	4	4	**0**
IIR filter iir()	3	3	3	4	1
IIR cascaded biqauds iircas4()	3	4	4	5	1
Lattice inv analysis latanal()	2	3	3	3	**0**
Lattice fwd synth. latsynth()	0	2	2	2	**0**
LMS adaptive filter lms()	1	3	3	3	**0**
All zero LMS lms_fir()	0	1	1	1	**0**
Max value max()	3	3	3	3	**0**
Max value index maxindex()	2	2	2	2	**0**
Min error search minerr()	4	9	9	11	2
16 tap all zero filter rb_fir1()	3	16	16	18	2
16 tap all zero filter rb_fir2()	9	6	9	9	**0**
Run length encoding run()	4	3	4	4	**0**
1 sample all zero filter	0	1	1	2	1
Vector max vec_max()	2	3	3	3	**0**
Vector multiply vec_mpy()	0	2	2	2	**0**

The input file is created by hand, and thus we are free to transform it by performing such optimizations as if-conversion, loop unrolling, and other transformations that eliminate or minimize loop-carried dependencies. A compiler can automate these techniques. We find the standalone tool to be valuable for experimentation since it allows us to focus specifically on the cluster assignment algorithm, but ultimately this algorithm would be implemented in a compiler.

The cluster assignment algorithm runs before an attempt at modulo scheduling a loop. The MII=MAX(RecMII, ResMII) for a loop is computed before and after cluster assignment. We measure how successful the algorithm is by measuring how close the PartMII comes to the MII. PartMII-MII=0 indicates that for the given loop kernel the algorithm found a PartMII at the MII. These results reinforce that our cluster assignment algorithm is an excellent technique for partitioning instructions to clusters without significantly increasing the MII.

5 Conclusion

Exploiting the fine-grain parallelism present at the machine instruction level, ILP processors have multiple functional units capable of executing in parallel. Because of physical limitations, functional units are partitioned into clusters with local register files. Instruction must be assigned to clusters in a way that minimizes inter-cluster transfers. Software pipelining is an instruction scheduling technique for exploiting ILP across loop iterations, and modulo scheduling is a method for implementing software pipelining.

We implemented a modulo scheduling aware cluster assignment algorithm that takes an architecture description and a loop kernel and outputs a cluster assignment. Using media and signal processing loop kernel benchmarks, we studied the effectiveness of our algorithm on a commercially available two-cluster VLIW processor. We presented the results on a set of forty DSP loop kernel benchmarks, and we defined a new measurement called the Partition MII or PartMII to quantify the impact of cluster assignment on the MII. Our results show that our algorithm is successful at assigning operations to clusters with a nominal average increase in the MII.

Acknowledgments. We would like to thank the reviewers for their valuable feedback on early versions of this paper.

References

1. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: 10th Symposium on Principles of Programming Languages, pp. 177--189. ACM, New York, NY, USA (1983)
2. Ellis, J.R.: Bulldog: A Compiler for VLIW Architectures. MIT Press, Cambridge, MA, USA (1986)

3. Hiser, J., Carr, S., Sweany, P.: Register assignment for software pipelining with partitioned register banks. In: 14th International Parallel and Distributed Processing Symposium, pp. 211--218. IEEE Computer Society, Los Alamitos, CA, USA (2000)
4. Lowney, P.G., Freudenberger, S.M., Karzes, T.J., Lichtenstein, W.D., Nix, R.P., O'Donnell, J.S., Ruttenburg, J.C.: The multiflow trace scheduling compiler. J. of Supercomputing, 7(1/2), 51--142 (1993)
5. Rau, B., Glaeser, C.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: 14th Annual Microprogramming Workshop, pp. 183-197. IEEE Computer Society (1981)
6. Rau, B.R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In 27th International Symposium on Microarchitecture, pp. 283-299, IEEE Computer Society, Los Alamitos, CA, USA (1994)
7. Fisher, J.A., Faraboschi, P., Young, C.: Embedded Computing: a VLIW Approach to Architecture, Compilers, and Tools. Morgan Kaufman Publishers, San Francisco, CA, USA (2005)
8. Krishnamurthy, G., Granston, E.A., Stotzer, E.J.: Affinity-based cluster assignment for unrolled loops. In: 16th International Conference on Supercomputing, pp.107--116. ACM, New York, NY, USA (2002)
9. Aleta, A., Codina, J.M., Sanchez, J., Gonzalez, A.: Graph-partitioning based instruction scheduling for clustered processors. In: 34th International Symposium on Microarchitecture, pp. 150--159. IEEE Computer Society, Los Alamitos, CA, USA (2001)
10. Capitanio, A., Dutt, N., Nicolau, A.: Partitioned register files for VLIWs: A preliminary analysis. In: 25th International Symposium on Microarchitecture, pp. 292--300. IEEE Computer Society, Los Alamitos, CA, USA (1992)
11. Codina, J.M., Sanchez, J., Gonzalez, A.: A unified modulo scheduling and register allocation technique for clustered processors. In: 10th International Conference on Parallel Architectures and Compilation Techniques, pp. 175--184. IEEE Computer Society, Los Alamitos, CA, USA (2001)
12. Kailas, K., Ebcioglu, K., Agrawala, A.: CARS: A new code generation framework for clustered ILP processors. In: 7th International Symposium on High-Performance Computer Architecture, pp. 133--146. IEEE Computer Society, Los Alamitos, CA, USA (2001)
13. Nystrom, E., Eichenberger, A.E.: Effective cluster assignment for modulo scheduling. In: 31st International Symposium on Microarchitecture, pp. 103--114. IEEE Computer Society, Los Alamitos, CA, USA (1998)
14. Özer, E., Banerjia, S., Conte, T.: Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In: 31st International Symposium on Microarchitecture, pp. 308--315. IEEE Computer Society, Los Alamitos, CA, USA (1998)
15. Stotzer, E., Leiss., E.: Modulo scheduling for the TMS320C6x VLIW DSP architecture. In: Workshop on Languages, Compilers and Tools for Embedded Systems, pp. 28--34. ACM, New York, NY, USA (1999)
16. Chu, M., Fan, K., Mahlke, S.: Region-based hierarchical operation partitioning for multicluster processors. In: Conference on Programming Language Design and Implementation, pp. 300-311. ACM, New York, NY, USA (2003)
17. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J: Software pipelining. ACM Computing Surveys, *27*(3), 367--432 (1995)
18. TMS320C6000 CPU and Instruction Set Reference Guide (SPRU189G). Texas Instruments, Inc., Dallas, TX, USA (2006)
19. Codina, J., Llosa, J., Gonzalez, A.: A comparative study of modulo scheduling techniques. In: International Conference on Supercomputing, pp. 97--106. ACM, New York, NY, USA (2002)