Register Pressure-Based Modulo Scheduling for Clustered VLIW Architectures

Alex Aletà[1], Josep M. Codina[1], Jesús Sánchez[1], Antonio González[1], and David Kaeli[2]

[1] Dept. of Computer Architecture, UPC, Barcelona, SPAIN
aaleta,jmcodina,fran,antonio@ac.upc.es
[2] Dept. of Electrical and Computer Eng., Computer Architecture Research Lab.
Northeastern University, Boston, MA, USA
kaeli@ece.neu.edu

Abstract. This paper presents a new modulo scheduling algorithm for clustered microarchitectures. The main feature of the proposed scheme is that the assignment of instructions to clusters is done by means of graph partitioning algorithms, and together with a pseudo-scheduling of the instructions. This pseudo-scheduling is a simplified version of the final scheduling step that basically estimates the main constraints that will determine the outcome of the final scheduling process. The final scheduling process is bi-directional and includes on-the-fly spill code generation. The proposed scheme is evaluated for the SPECfp95 benchmark suite and compared with previous approaches. Results show that better schedulings are abtained for almost all programs and different architectures. For some processor configurations the average speedup is about 20

1 Introduction

We are presently seeing rapid growth in the embedded and low-power processor domains. A number of recent processors use a clustered microarchitecture that physically partitions functional elements and resources. The components of each cluster are simpler and thus are faster and consume less power than more unified designs. Cluster components can be laid out close together, which can reduce signal transmission delays [13]. Long, slow wires are used to interconnect clusters. The use of clustering is especially noticeable in the DSP market, including Analog Devices' TigerSHARC [12], BOPS's ManArray[24], HP/ST's Lx [9], and the Equator MAP1000 [21]. All of these processors implement a VLIW architecture, and rely on the compiler to perform instruction scheduling.

The compiler plays a critical role in the success of a clustered VLIW processor. The compiler must smartly schedule code to make best use out of the multiple resources provided. In this paper we focus on instruction scheduling for clustered processors. We limit our focus to scheduling software pipelined loops [19], since a majority of the execution on this class of processor is found in loop bodies. We propose a new modulo scheduling algorithm for clustered architectures with distributed register file and functional units.

A critical step of the scheduling process is the assignment of instructions to clusters, since this will determine the penalties caused by inter-cluster communications. Another important objective is balance the workload (instructions,

register pressure, etc.) among clusters. For this purpose, we propose an approach that performs the cluster assignment almost at the same time as the instruction scheduling. Previous works have shown that performing this two steps at the same time is beneficial. However, since cluster assignment is usually based on trying many alternatives, doing a complete schedule for each of them may be unaffordable. Instead, the proposed scheme just performs a pseudo-scheduling that basically estimates the main constraints that will determine the outcome of the final scheduling process. The final scheduling process is bi-directional and includes on-the-fly spill code generation.

The proposed scheme is evaluated for 678 different loops taken from the SPEC95fp benchmark suite, which represent around 95total execution time of these programs. The results for different configurations show that this new scheme outperforms previously proposed techniques for all benchmarks and all the architectures.

The remainder of this paper is organized as follows. Section 2 provides an overview of clustered VLIW microarchitectures and modulo scheduling. Section 3 describes the proposed scheduling scheme. Section 4 reports on the performance and compares it with previous proposals. Section 5 outlines a number of works. Section 6 summarizes the work and describes directions for further improvement.

2 Background

In this section we describe the assumed microarchitecture and review the main concepts of modulo scheduling.

2.1 Microarchitecture

Centralized resources tend to increase design complexity and limit the scalability of a design. Clustered VLIW architectures decentralize some components. A single cluster is composed of multiple functional units sharing a common register file. We consider three types of functional units: 1) integer arithmetic, 2) floating-point arithmetic and 3) memory access. Multiple clusters share a common memory hierarchy and communicate operands among clusters using a set of dedicated *register buses*. The ISA includes instructions that read a value from a register in one cluster and copy it into another register of a different cluster. For the sake of simplicity, we assume homogeneous clusters although the proposed algorithms can easily be generalized for heterogeneous clusters.

Figure 1 shows the assumend microarchitecture. VLIW instructions are issued to each cluster in a lockstep fashion (all clusters work on the same VLIW instruction together). In each cycle, every cluster will fetch the operations contained in their corresponding part of a VLIW instruction.

2.2 Modulo Scheduling

Modulo scheduling is an instruction scheduling technique for program loops [19] ...reference Rau... . It has been shown to be a very effective technique for exploiting the available parallelism in cyclic codes. Modulo scheduling attempts to reduce the *Initiation Interval* (II) associated with a loop (the II is a measure of the number of cycles between successive iterations of loop), while respecting

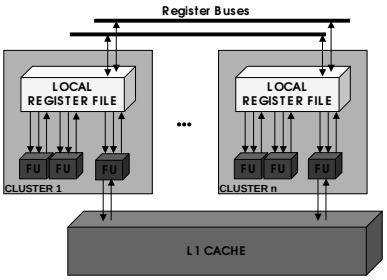

Fig. 1. Clustered VLIW microarchitecture. Register values are communicated through intercluster register buses.

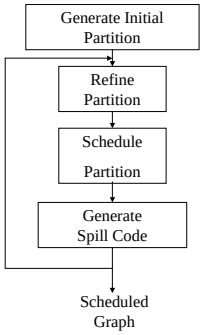

Fig. 2. The high-level structure of our scheduling framework.

data dependences and resource requirements. For loops with high trip counts, the II can be used to approximate the overall runtime of the loop.

High register bus pressure caused inter-cluster communications and high register pressure (i.e., many operands live concurrently) can dramatically increase the II [20].

In this work we look to provide an scheduling approach that addresses all the above issues: maximize parallelism and minimize register pressure and register bus pressure.

Modulo scheduling uses a Data Dependence Graph (DDG) to represent the relationships between different operations in a loop. The set of nodes (V) represents the set of instructions and the set of edges (E) represents the dependences among these instructions. The problem of assigning instructions to clusters can be stated as a graph partitioning problem. Each subgraph of the resulting partition includes the instructions that will be executed in a concrete cluster.

3 Proposed Algorithm

Figure 2 provides the high-level flow of the proposed algorithm. First of all, an initial partition is computed taking into account the minimum II *MII*, an estimation of the register pressure and the pressure on the register buses, and the resource contraints for each cluster.

Then, the instructions are scheduled according to the computed partition one at a time. If an instruction cannot be scheduled in the predetermined clusters, the other clusters are tried. If an instruction cannot be scheduled in any cluster, the II is increased, the partition is modified and the scheduling process is repeated again.

These steps are further detailed below.

3.1 Generate Initial Partition

The initial partition is computed through a multilevel partitioning strategy [], which has been shown to be very effective. A multilevel partitioning strategy works as follows.

First, a preliminary partition is generated by coarsening the nodes of the DDG. Coarsening consists of choosing several disjoint sets of nodes of a graph and consolidating all the nodes of a set into a single coarser node. This results in a new graph with fewer nodes, each of them representing several instructions. Coarsening is repeated iteratively until a graph containing as many nodes as the number of intended partitions (i.e., the number of clusters) is obtained. This graph represents the preliminary partition, in which all instructions of a coarse node are assigned to a given cluster.

Then, this preliminary partition is enhanced by walking back through all the intermediate coarse graphs in reverse order. For each graph, some nodes are moved from one cluster to another if this improves a certain figure of merit.

The second step in a multilevel partitioning strategy simply induces back the partition of the smaller graphs into the bigger ones by assignig every node of the bigger graph to the same cluster where the macro-node of the smaller graph to which it is collapsed. Finally, the partition of the bigger graph is improved using heuristics to move nodes among clusters.

Graph Coarsening The coarsening process involves performing a matching in the current graph. For a graph $G = (V, E)$, a matching is a set M of the edges in E such that each node can be connected to at most one of the selected edges. Each edge of the graph is weighted using two values: 1) the impact on the schedule if we increase the delay on this edge, and 2) the *slack* time, which represents the number of cycles that could be added to this edge without affecting execution time. At each step of the coarsening we select the maximum weight matching. The pair of nodes connected by each selected edge are then fused into a single coarser node.

Enhancing the Prelimirany Partition A number of heuristics have been proposed for improving a partition, most of them based on the work of Kernighan and Lin [18] and the improvements by Fiduccia and Mattheyes [11]. The general idea in these algorithms is to move nodes from one subgraph to another until no further improvement can be achieved. Nevertheless, in our case it is not easy to decide whether a movement improves a partition. Our goal is to achieve a partition that can be scheduled minimizing the execution time of the loop. For this purpose, multiple constraints have to be taken into account such as recurrences with edges between instructions in different clusters, register pressure, bus pressure, length of the scheduling, etc. Much of this information is available only at the scheduling time so partitioning the graph before scheduling could lead to bad decisions. On the other hand, scheduling is done node by node so it is difficult to make good global decisions at scheduling time. Therefore, a unified scheme that performs cluster assignment and scheduling at the same time may be the most effective approach. Since building a real schedule is quite an expensive task, the proposed scheme is based on exploring the solution space for partitioning, but instead of computing a detailed schedule for each possibility, the decisions are based on estimating the final schedule in a simple way as described below.

Aproximate Schedule To work out the aproximate schedule, we first compute a lower bound of the II for the current partition taking into account bus

pressure and recurrences that span multiple clusters: $II_{lowerbound} = max(II_{res}, II_{rec'}, II_{bus})$ $II_{bus} = [intsuppartof(ncoms/nbuses)] * latbus$ where n coms is the number of communications necessary to schedule the partition, n buses is the number of buses in the architecture and lat bus is the latency of the buses. To compute $II_{rec'}$ we proceed as in ...reference... but taking into account the latency of the edges between instructions in different clusters.

Then, assuming $II = II_{lowerbound}$ we try to find a suitable slot for each node. Since the aproximate schedule has to be as similar as possible to the real one, nodes are scheduled following the same rules as the real scheduler does. Therefore, according to the ordering of the SMS ...reference... each node is scheduled as close as possible to his predecessors/successors in order not to enlarge lifetimes. Unlike the real scheduler, when there is not any available slot to schedule an instruction in the cluster to which it belongs, we assign that node to a given cycle even if resources are not available at this cycle. This cycle is the earliest start ...ref... minus one if it has only predecessors in the partial schedule, the latest start plus one if it has only successors and the midpoint between the earliest start and latest start otherwise. This +/- 1 cycle is intented to penalize this partition by enlarging the lifetimes and the lenght of the schedule. This penalty is relatively small since this is just an intermediate partition that can still be improved by further enhancement steps. On the other hand, if the node that cannot be scheduled belongs to a recurrence, a much bigger penalty is considered since introducing an additional delay in a recurrence has usually a significant impact on performance. In particular, it is assumed that for this partition $II = II_{lowerbound} + 2 * latbus + 1$.

Note that for this aproximate schedule, an unlimited number of registers is assumed. However, once the approximate schedule is computed, the lifetimes required by it, as well as the maximum number of lifetimes that overlap (which is usually refer to as MaxLive) are computed. These paramenters are later used by the register pressure is not taken into account during that aproximate scheduling. However, we can easily count the number of lifetimes of each cluster (see the algorithm in Figure 4) Proceeding in this way, all nodes are pseudo-scheduled (assigned to a cycle) searching no more than $II_{lowerbound}$ different positions. Thus, the computing time of that aproximate scheduling is linear with $II * |V|$.

Enhancing Heuristics In order to determine which movements of nodes from one cluster to another enhance the partition, two different steps are applied. First, any excess workload in a cluster (i.e., when the instructions assigned to a cluster require more resources than those available) is tried to be moved to another cluster with empty slots. Then, we consider inter-cluster node movements that do not cause any workload excess and reduce the execution time. Figure 3 shows a pseudocode for these enhancing heuristics. They are further detailed below.

1. **Workload Balancing** - According to the initiation interval (II) and the resources available in the architecture, there are limited slots to schedule instructions in each cluster. If the usage of a resource (registers, memory or functional units (FUs)) in a cluster is too high then we will try to move

nodes that use this resource to other clusters where the load of this resource is lower. Note, we only consider memory and FUs during the enhancing step since once the partition is balanced[1] movements overloading a cluster are not allowed.

2. **Reducing Execution Time** - After improving the workload balance, we look for a modified partition that is likely to reduce the execution time of the loop. For this purpose, precise information on the causes that could increase execution time are required in order to guide this enhancing step. This information is obtained from the aproximate schedule (see above). This proceeds as follows: first of all, nodes are moved, one at a time, to adjacent clusters (a node is adjacent to a cluster if any of its predecessors/successors is assigned to that cluster). Then, the aproximate schedule of every resulting partition is worked out. Finally, all the aproximate schedules are compared, the best one is selected and the movement that induced it is done. The best aproximate schedule is the one that minimizes execution time (that is, $Texec = II * Niter + lengthsched$) where II and $lengthsched$ (lenght of the schedule) are the estimations obtained from the aproximate schedule. In case of a tie, the one that minimizes the register and the bus pressure is chosen. That is, if $II_{bus} > II$ the one that minimizes the number of communications; otherwise, the one that minimizes the cluster with the highest total number of accumulated lifetime slots (according to the aproximate schedule) is chosen. In case of a tie, other heuristics are used which are not described for the shake of simplicity. If moving a node from one cluster to another overloads the second cluster (i.e. the latter cluster has not enough resources) we look for a node in the second cluster such that moving it to the first one balances the partition again. The aproximate schedule is built with the two movements done. No movement overloading a cluster is considered.

Previous work [1] that used graph partitioning algorithms did not compute aproximate schedules during partitioning. Then, the partitioning decisions were made with much less relevant information. Instead, the partitioning algorithm we present in this work has very precise information coming from the aproximate schedule. Therefore, moving nodes among clusters during the partitioning step can target different goals, depending on the most constraining factors: reducing the number of communications, spreading register pressure, better splitting recurrences among clusters or reducing the length of the schedule. Moreover, the aproximate schedule is quite accurate, specially when the partition is balanced and there is enough space to schedule all instructions in the cluster they have been assigned to by the partition. We will use the results presented in [1] as our baseline scheme, since they represent a state-of-the-art approach to modulo scheduling loops on a clustered VLIW processor.

3.2 Scheduling the Instructions

Once a partition has been computed, each instruction is scheduled in the assigned cluster. This scheduling is a bidirectional scheme borrowed from the approach

[1] In our context, balanced means that there are enough resources in each cluster to schedule all the operations assigned ot them.

```
BEGIN Best_Movement:
 While (Pressure too high on FUs/memory and
        not every move attempted without improvement)
    {Foreach cluster
       { Foreach node
            { Try to move any node vi from cluster Cj to any other cluster;
              Compute the resulting FU pressure;
            }
       }
     Pick the best movement;
     Update the partition;
    }

 While (Pressure too high on registers and
        not every move attempted without improvement)
    {Foreach cluster
       { Foreach node
           { Try to move any node vi from cluster Cj to any other cluster;
             Compute the resulting register pressure;
           }
       }
     Pick the best movement;
     Update the partition;
    }
END Best_Movement


BEGIN Best_Adjacent_Movement:
 While (Pressure too high on FUs and
        not every move attempted without improvement)
    {Foreach cluster
        { Foreach node
            { Try to move any node vi from cluster Cj to all other clusters;
              Compute the resulting FU pressure;
            }
        }
     Pick the best movement;
     Update the partition;
    }

 While (Improvement is found)
    {Forall nodes v adjacent to the cut of the partition
       { Try to move vi from cluster Ci to the adjacent cluster Cj;
         If not beneficial
            { Identify adjacent node u in Cj to move to Ci;
            }
       }
     Pick the best movement;
     Update the partition;
    }
END Best_Adjacent_Movement
```

Fig. 3. Pseudocode for our two refinement steps.

```
Foreach clusters
  {sum = 0;
   Foreach node
     { Find the longest lifetime for this node
        (i.e., the longest edge, measured in cycles);
       sum = sum + longest lifetime;
     }
   IIreg(cluster) = sum / # of regs per cluster;
  }
Select the largest IIreg(cluster).
```

Fig. 4. Pseudocode for our register pressure estimation.

Architecture	Clusters	Regs	Register Bus Lat
Arch I	2	64	1
Arch II	4	64	1
Arch III	4	64	2
Arch IV	2	32	1
Arch V	4	32	1

Table 1. Configurations considered in this work.

Resource	Unified	2-cluster	4-cluster
INT/cluster	4	2	1
FP/cluster	4	2	1
MEM/cluster	4	2	1

Table 2. Clustered VLIW resource configurations.

Latency	INT	FP
MEM	2	2
ARITH	1	3
MUL/ABS	2	6
DIV/SQR/TRG	6	18

Table 3. Operation latencies.

presented in URACAM [4], which was shown to be very effective in terms of exploiting parallelism and reducing register pressure. URACAM scheduler gives priority to nodes according to their criticality and tries to avoid stretching lifetimes. Besides, it tries to keep balanced the usage of all critical resources.

Whenenver an instruction cannot be scheduled in the cluster assigned by the partition, the other clusters are tried. If no other alternative cluster is feasible, then the II is increased and the partition is refined before trying again to schedule all the instruction.

3.3 Refine the Partition

Once the II is increased, there appear extra slots in each clusters. The refining step starts from the previous partition and tries to optimize it using the extra slots. First of all, the subgraphs that corresponds to the nodes in each cluster are coarsened following the same algorithm as the one described in section ...X.... Then, the partition is enhanced by using the 'Reducing Execution Time' step of the Enhancing phase described in section ...X...

4 Evaluation

Our algorithms have been implemented using the ICTINEO compiler framework [2]. We evaluate 678 loops present in the SPECfp95 benchmark suite. We study five different architectures, which are defined in Table 1. All architectures assume a single intercluster register bus. Table 2 lists the functional unit resources provided in each cluster. Table 3 provides functional unit latencies. We report on machine configurations which vary parameters related to these design features. We assume a perfect memory system in this work, though plan to consider memory effects in future work (since this impacts the cost of a spill).

For each of these configurations, we present results for our three different scheduling algorithms:

- Baseline - Results reported in [1], which utilized the same compiler infrastructure, but utilize the URACAM [4] scheduler to produce the final schedule without considering register pressure when partitioning the graph.

- No Dupe - Application of the new algorithm without duplicating nodes, and
- Dupe - Duplication of nodes in other clusters to reduce pressure on the intercluster bus. This result includes our two spill code analyzes.

We also include results for a unified cluster comprised of four functional units of each type and a unified register file.

Our performance metric is the instructions per cycle (IPC). Note that this metric does not consider the impact of the cycle time, which is one of the important benefits of clustering. Thus, the IPC of an equivalent non-clustered architecture can be considered as a lower-bound for the clustered organizations since no communication is needed.

In Figures 5, 6 and 7 we present IPC numbers for the three configurations with 64 registers (ArchI, ArchII and ArchIII respectively). We can see that for all programs except swim (in ArchI with spills) we obtain better results than the baseline. The ArchI (2 clusters, 64 registers, 1 register bus with 1 cycle latency) represents the least restrictive configuration considered (with respect to an equivalent unified architecture with the same number of resources and registers, but without paying any communication overhead). As we can see from Figure 5, the algorithm proposed in this paper obtains IPC results very close to the unified architecture. In fact, the degradation in IPC is 3% on average. From more restrictive configurations (ArchII and ArchIII) the degradation, as expected, increases, but on average we obtain an improvement of about 10%-20% when compared with the baseline algorithm. From these graphs we can also see that, except for particularly cases (swim and turb3d in ArchI, and swim in ArchII), the code duplication optimization improves (or does not degrade) the proposed algorithm. On average, for the three configurations a 2-3% improvement is obtained. Note, however, that for some programs (e.g., turb3d in ArchI, tomcatv, turb3d and fppp in ArchII, and tomcatv and swim in ArchIII) this improvement in more noticeable.

In Figures 8 and 9 we show the results for the two configurations with 32 registers (ArchIV and ArchV). As we saw in configurations with 64 registers, we generally outperform the baseline algorithm, although for some programs (swim and apsi in ArchIV and apsi and fpppp in ArchV) we get particular poorer results.

5 Related Work

Finding an optimal schedule in a resource constrained environment is an NP-complete problem. For this reason, many heuristics have been proposed in order to find near-optimal schedules. These heuristics have different goals: increasing throughput [15, 25], minimizing register pressure [7, 6], reducing the effect of the cache misses, or improving several of them simultaneously [14, 6, 26, 20]. All of these studies focus on modulo scheduling algorithms targeting unified (i.e., non-partitioned) architectures.

There are several works related to acyclic code scheduling for clustered architectures [8, 3, 5, 16, 23]. The most closely related work to our ideas include the work of Kailas, Ebcioglu and Agrawala [17] where they proposed an approach

to cluster assignment, instruction scheduling and register allocation on a single phase, based on a list scheduling scheme. These works differ from the approach presented in this paper in that they target instruction scheduling for acyclic code. Besides, they also use different cluster assignment heuristics. Some modulo scheduling approaches, targeting clustered VLIW architectures, have been recently proposed:

- Nystrom and Eichenberger [22] investigated cluster assignment for modulo scheduling, mainly focusing on minimizing execution overhead due to intercluster communication with a two-step approach: first partitioning the dependence graph of the loop body (assigning each operation to a cluster), and then scheduling the operations following the graph partition.
- Fernandes et al. [10] proposed a modulo scheduling approach integrating scheduling and cluster assignment in a single step. However, they assume an architecture with an unusual register file organization based on a set of local queues for each cluster and a queue file for each communication channel.
- Sánchez and González [28] proposed a unified assign-and-schedule approach in which cluster selection and scheduling are done in a single phase. That work was later extended to deal with a distributed cache memory [27].

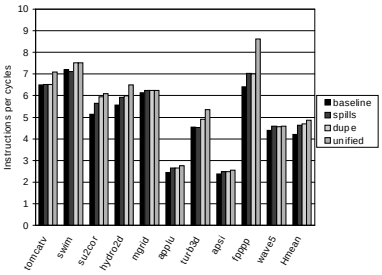

Fig. 5. IPC numbers for an architecture with 2 clusters, 64 registers, 1 register bus with a 1 cycle register bus latency (Arch I)

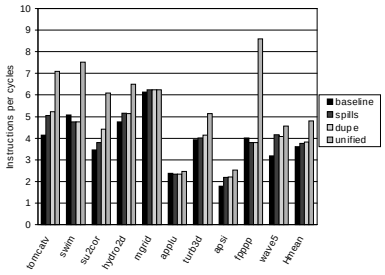

Fig. 8. IPC numbers for an architecture with 2 clusters, 32 registers, 1 register bus with a 1 cycle register bus latency (Arch IV)

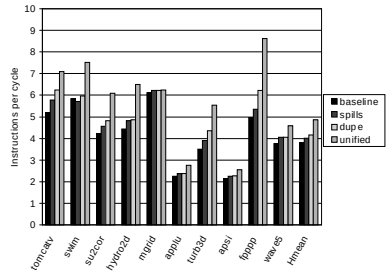

Fig. 6. IPC numbers for an architecture with 4 clusters, 64 registers, 1 register bus with a 1 cycle register bus latency (Arch II)

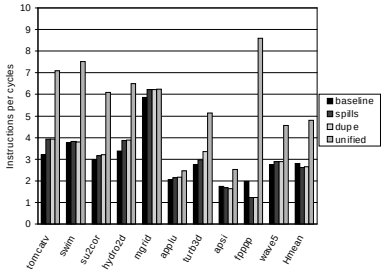

Fig. 9. IPC numbers for an architecture with 4 clusters, 32 registers, 1 register bus with a 1 cycle register bus latency (Arch V)

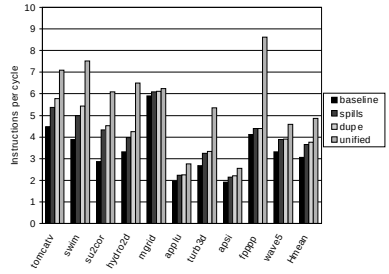

Fig. 7. IPC numbers for an architecture with 4 clusters, 64 registers, 1 register bus with a 2 cycle register bus latency (Arch III)

- Codina et al. [4] presented an approach to deal with instruction scheduling, cluster assignment and register allocation in a single phase with a smart approach to insert spill code on-the-fly and effective mechanisms to deal with communications, register and memory pressure at the same time.
- Zalamea et al. [30] also proposed a technique to cluster assignment, instruction scheduling and register allocation based on an iterative scheme [25] with some heuristics to deal with spill code [29].
- Aletà et al. [1] presented a graph-partitioning based approach with close interaction to the scheduling phase. The main goal was to improve the results obtained for a technique that combines cluster assignment, instruction scheduling and register allocation in single phase [4] with the global view of the whole problem given by a technique based on a partitioning of graph. This scheme has been used as the baseline for comparison with the proposal of this paper.

6 Summary

We have presented a new modulo scheduling algorithm for clustered processors. The main novelty is the use of a unified phase that does the partioning with a quite accurate (but not totally precise) estimation of the complete schedule of the instructions. This unified approach allows for a much more effective partition.

We have compared the proposed scheme with a state-of-the-art approach that is based on graph partitioning algoriths like the proposed approach. Results show that the use of an accurate estimation of the schedule to take partitioning decisions significantly improves the performance. In average, the proposed scheme is ..... than .... and for some programs such as ... the improvement is ....

Acknowledgments

This work has been partially supported by the ESPRIT project MHAOTEU (EP 24942), the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01, Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00664 UPC APTIND and Analog Devices. David Kaeli is supported by the *Ministry of Education, Culture and Sports* of Spain and the National Science Foundation.

References

1. A. Aletà, J. M. Codina, J. Sánchez, and A. Gónzalez. Graph-Partitioning Based Instruction Scheduling for Clustered Processors. In *Proc. of 34th Int. Symp. on Microarchitecture*, Dec 2001.
2. E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera, and M. Valero. Ictineo: A Tool for Research on ILP. In *Supercomputing 96*, 1996.
3. A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register File for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proc. of the 25th Int. Symposium on Microarchitecture*, pages 292–300, 1992.
4. J. M. Codina, J. Sánchez, and A. Gónzalez. A Unified Modulo Scheduling and Register Allocation Technique for Cluster Processors. In *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 175–184, Sept 2001.

5. G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers. Technical Report HP-98-13, HP Labs Technical Report, Jan 1998.
6. A. Eichenberger and E. Davidson. Stage Scheduling: A Technique to Reduce the Register Requirements of a Module Schedule. In *Proc. of the 28th Int. Symposium on Microarchitecure*, pages 338–349, 1995.
7. A. Eichenberger, E. Davidson, and S. Abraham. Optimum Module Schedules for Minimum Register Requirements. In *Proc. of Supercomputing '95*, 1995.
8. J. Ellis. *Bulldog: A Compiler for VLIW Architecture*. MIT Press, Cambridge, MA, 1986.
9. P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proc. of the 27th Int. Symp. on Computer Architecture*, pages 203–213, June 2000.
10. M. Fernandes, J. Llosa, and N. Topham. Partitioned Schedules for Clustered VLIW Architectures. In *Proc., 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'1998)*, pages 386–391, March 1998.
11. C. Fiduccia and R. Mattheyes. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. of 19th Design Automation Conference*, pages 175–181, 1982.
12. J. Fridman and Z. Greenfield. The TigerSharc DSP Architecture. *IEEE Micro*, pages 66–76, Jan-Feb 2000.
13. R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proc. of the IEEE*, pages 490–504, April 2001.
14. R. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proc. of the Int. Conf. on Programming Languages, Design and Implementation*, pages 318–328, 1993.
15. S. Jain. Circular Scheduling: A New Technique to Perform Software Pipelining. In *Proc. of the Int. Conf. on Programming Languages, Design and Implementation*, pages 219–228, 1991.
16. S. Jang, S. Carr, P. Sweany, and D. Kuras. A Code Generation Framework for VLIW Architectures. In *Proc. of the 3rd Int. Conf. on Massively Parallel Computing Systems*, April 1998.
17. K. Kailas, K. Ebcioglu, and A. Agrawala. CARS: A New Code Generation Framework for Clusterd ILP Processors. In *Proc. of the 7th Int. Symposium on High Performance Computer Architecture*, pages 133–143, 2001.
18. B. Kernighan and S. Lin. An Effective Heuristic Procedure for Partitioning Graphs. *Bell Syst. Tech. Journal*, pages 291–307, 1970.
19. M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the 8th Int. Conf. on Programming Languages, Design and Implementation*, pages 258–267, June 1988.
20. J. Llosa, M. Valero, E. Ayguadé, and A. González. Modulo Scheduling with Reduced Register Pressure. *IEEE Transactions on Computers*, 47(6):625–638, 1998.
21. MAP1000. MAP1000 unfolds at Equator. *Microprocessor Report*, 12(16), Dec 1998.
22. E. Nystrom and A. E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *Proc. of the 31st Int. Symposium on Microarchitecture*, 1998.
23. E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proc. of the 31st Int. Symposium on Microarchitecture*, pages 308–315, 1998.
24. G. Pechanek and S. Vassiliadis. The ManArray Embedded Processor Architecture. In *Proc. of 26th Euromicro Conference*, pages 348–355, Sept 2000.
25. B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of 27th Int. Symp. on Microarchitecture*, pages 67–74, Nov 1994.

26. J. Sánchez and A. González. Cache Sensitive Modulo Scheduling. In *Proc. of 30th Int. Symp. on Microarchitecture*, pages 338–348, Dec 1997.
27. J. Sánchez and A. González. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. In *Proc. of the 33rd Int. Symposium on Microarchitecture*, pages 124–133, Dec 2000.
28. J. Sánchez and A. González. The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures. In *Procs. of the Int. Conf. on Parallel Processing (ICPP'00)*, pages 555–562, August 2000.
29. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved Spill Code Generation for Software Pipelined Loops. In *Procs. of the Programming Languages Design and Implementation (PLDI'00)*, June 2000.
30. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. In *Proc. of the 34th Int. Symp. on Microarchitecture*, December 2001.