Exploiting Pseudo-schedules to Guide Data Dependence Graph Partitioning

Alex Aletà[1], Josep M. Codina[1]
Jesús Sánchez[1,2], Antonio González[1,2]
David Kaeli[3]
Dept. of Computer Architecture, UPC, Barcelona, SPAIN[1]
Intel Barcelona Research Center, Intel Labs, Barcelona, SPAIN[2]
Northeastern University, Boston, MA, USA[3]
email: aaleta,jmcodina,fran,antonio,dkaeli@ac.upc.es

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 that are guided by a pseudo-scheduler. This pseudo-scheduler is a simplified version of the full instruction scheduler and estimates key constraints that would be encountered in the final schedule.

The final scheduling process is bi-directional and includes on-the-fly spill code generation. The proposed scheme is evaluated against previous scheduling approaches using the SPECfp95 benchmark suite. Our modeling results show that better schedules are obtained for most programs across a range of different architectures. For a 4-cluster VLIW architecture with 32 registers and a 2-cycle inter-cluster communication delay we obtain an average speedup of 38.5%.

1. Introduction

We are presently seeing rapid growth in the embedded and low-power processor domains. A number of these 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 energy than more unified designs. Cluster components can be laid out close together, which can reduce signal transmission delays [18]. Long, slow wires are used to interconnect clusters. The use of clustering is especially noticeable in the DSP market, including Analog Devices' TigerSHARC [14], BOPS's ManArray[31], HP/ST's Lx [11], and the Equator MAP1000 [27]. 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 carefully schedule code to make best use of the multiple resources provided. In this paper we study instruction scheduling for clustered processors. We limit our focus to scheduling software pipelined loops [4], since a majority of the execution on this class of processors is found in loop bodies. We propose a new modulo scheduling algorithm for clustered architectures with a distributed register file and functional units.

One major goal of any scheduling process targeting a clustered architecture is to properly assign instructions to particular clusters, since this will determine the amount of latency introduced by inter-cluster communications. A second objective of the scheduler is to carefully balance the workload (instructions, register pressure, etc.) across all clusters. For this purpose, we propose an approach that performs cluster assignment while considering instruction scheduling. Previous work has shown that performing these two objectives concurrently can be beneficial. However, since cluster assignment typically considers many alternative assignments, computing a complete schedule for each alternative assignment would be expensive (computationally). Instead, the proposed scheme computes a pseudo-schedule, estimating the key constraints that will greatly influence the outcome of the scheduling process.

The proposed scheme is evaluated for 678 different loops taken from the SPECfp95 benchmark suite, which represent around 95% of the total execution time of these programs. The results for different configurations show that this new scheme outperforms previously proposed techniques for most cases in all benchmarks and for a range of clustered architectures.

Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)
1089-795X/02 $17.00 © 2002 IEEE

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 reviews a number of related 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 assumed microarchitecture. VLIW instructions are issued to each cluster in a lockstep fashion (all clusters work on the same VLIW instruction together). During each cycle, every cluster will fetch the operations contained in their corresponding part of a VLIW instruction. A full description of the VLIW instruction set used in this work can be found in [36].

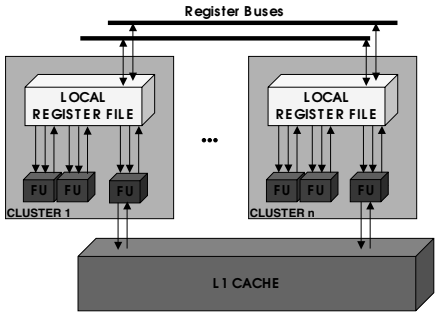

Figure 1. Clustered VLIW microarchitecture. Register values are communicated through inter-cluster register buses.

2.2. Modulo Scheduling

Modulo scheduling is an instruction scheduling technique for program loops [24, 33]. 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 a loop), while respecting data dependencies 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 by inter-cluster communications and high register pressure (i.e., many operands live concurrently) can dramatically increase the II [26]. In this work we look to provide an scheduling approach that

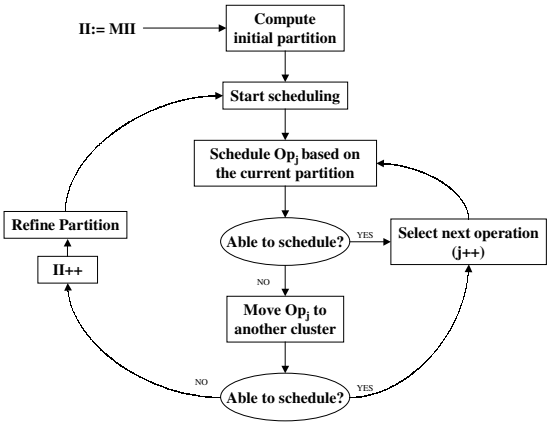

Figure 2. The high-level structure of our scheduling framework.

addresses all the above issues, exposing instruction-level parallelism while reducing register pressure, register bus pressure, and functional unit 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 that 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 initiation interval (*MII*), an estimation of the register pressure, the pressure on the register buses, and the resource constraints for each cluster. MII is the maximum between II_{res} (the initiation interval due to resources) and II_{rec} (the initiation interval due to recurrences); these two values are computed as in [32].

Then, instructions are scheduled according to their computed cluster assignment. If an instruction cannot be scheduled in the assigned cluster, the instruction is moved to a different cluster. If an instruction cannot be scheduled in any cluster, the II is increased, the partition is modified and the scheduling process is restarted. We describe these steps in more detail next.

3.1. Computing an Initial Partition

The initial partition is computed through a multi-level partitioning strategy [17], which has been shown to be very effective. Our multi-level partitioning strategy works as follows:

1. First, a preliminary partition is generated by coarsening the nodes of the DDG. Coarsening consists of choosing pairs of nodes in the graph and merging these nodes into a single coarser node. This produces a new graph with fewer nodes, with each coarsened node representing multiple 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.

2. Then, this preliminary partition is enhanced by walking back through all the intermediate graphs in reverse order. For each graph, some nodes are moved from one cluster to another, but only if the move improves a particular figure of merit (e.g., register pressure, functional unit pressure).

We describe each of these steps in more detail below.

3.1.1 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 [19], which represents the number of cycles that could be added to this edge without affecting execution time. At each step of the coarsening process we select the *maximum weight matching*, which was first described by Gabow in [15], and implemented in our work as described in [28]. The pair of nodes connected by each selected edge are then fused into a single coarser node.

3.1.2 Enhancing the Preliminary Partition

A number of heuristics have been proposed for improving a partition, most of them based on the work of Kernighan and Lin [23] and the improvements provided by Fiduccia and Mattheyes [13]. The general idea in these algorithms is to move nodes from one subgraph to another until no further improvement can be achieved. For our problem it is not always easy to decide whether a movement improves a partition. Our goal is to produce 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 schedule, etc. Much of this information is available only at scheduling time, so partitioning the graph before scheduling could lead to bad decisions. On the other hand, scheduling is done node by node, such that it is difficult to make good global decisions at scheduling time. Therefore, a scheme that performs cluster assignment and scheduling at the same time may be the most effective approach. Since building a complete schedule is quite an expensive task, the proposed scheme is based on exploring the solution space for partitioning, but is guided by a a simplified estimation we call a *pseudo-schedule*, as we describe next.

3.1.3 Computing a Pseudo-Schedule

To produce a pseudo-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:

$$II_{lowerbound} = max(II_{res}, II_{rec'}, II_{bus}), \text{ where}$$
$$II_{bus} = \lceil ncoms/nbuses \rceil * buslat, \text{ and}$$

where $ncoms$ is the number of communications necessary to schedule the partition, $nbuses$ is the number of buses in the architecture and $buslat$ is the latency of the buses. To compute $II_{rec'}$, we proceed as in [32], but also take 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 pseudo-schedule needs to be computed as accurately as possible, nodes are scheduled using the same rules used by the *full scheduler* [1]. Therefore, according to the ordering of the Swing Modulo Scheduler [25], each node is scheduled as close as possible to its predecessors/successors in order not to increase lifetimes. Unlike the full scheduler, if we do not find any slot available 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 in this cycle. We determine this cycle as follows [2]:

1. the earliest start [32] minus one if it has only predecessors in the partial schedule,
2. the latest start plus one if it has only successors, or otherwise
3. the midpoint between the earliest start and latest start.

The intent of decrementing/incrementing the earliest/latest start time by one is to penalize this partition by extending the lifetimes and the length of the schedule. This penalty is relatively small since this is just an intermediate partition that can still be improved upon further by later steps. For case 3, if the node that cannot be scheduled is the last instruction in a recurrence, a much larger penalty is assessed since introducing an additional delay in a recurrence usually imposes a significant penalty on performance. In particular, it is assumed that for this partition:

$$II = II_{lowerbound} + 2 * buslat + 1.$$

Since splitting a recurrence generally incurs two communications, we penalize the II by 2 * buslat. Note that for this approximate 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 is computed (see the algorithm in Figure 4). These parameters are later used to guide the heuristics.

[1] The full scheduler refers to the process of producing a schedule while respecting all constraints.

[2] Since the pseudo-schedule is not a real schedule, data dependences may not be respected.

Proceeding in this way, all nodes are pseudo-scheduled (assigned to a cycle), searching no more than $II_{lowerbound}$ different positions. Thus, the compute time for producing a pseudo-schedule is linear with $II * |V|$.

3.1.4 Enhancing Heuristics

In order to determine which node movements between clusters will be beneficial, two different optimizations are attempted. First, if we find any excess workload in a cluster (i.e., when the instructions assigned to a cluster require more resources than those available), we try to move the workload to another cluster that has empty slots. Next, 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 two enhancing heuristics. They are further described 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 not balanced [3], then we will try to move nodes that use this resource to other clusters where the load on this resource is lower.

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 pseudo-scheduler (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, a pseudo-schedule for every resulting partition is computed. Finally, all the pseudo-schedules are compared, the best one is selected and the movement that induced it is used. The best pseudo-schedule is the one that minimizes execution time (that is, $Texec = II * Niter + lengthsched$), where II and $lengthsched$ (length of the schedule) are estimations obtained from the pseudo-schedule, and Niter(the number of iterations) is obtained through profiling. In the case of a tie, the one that minimizes the register pressure and the bus pressure is chosen. That is, if $II_{bus} > II$ the one that minimizes the number of communications is chosen; otherwise, the one that minimizes the cluster with the highest total number of ac-

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

```
BEGIN Workload_Balancing:
 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 Workload_Balancing


BEGIN Reducing_Execution_Time:
 While (Not every move attempted without improvement)
    {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 Reducing_Execution_Time
```

Figure 3. Pseudocode for the two enhancing heuristics.

cumulated lifetime slots (according to the approximate schedule) is chosen.

When applying the above enhancement, if moving a node from one cluster to another overloads the second cluster (i.e. the latter cluster does not have sufficient resources), we look for a node in the second cluster such that moving it to the first cluster re-balances the partition.

In previous work [1], we proposed graph partitioning algorithms but did not provide approximate schedules to guide partitioning. As a result, the partitioning decisions were made with much less relevant information. In this work, the partitioning algorithms we present have very precise information produced by the pseudo-scheduler. 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 computed pseudo-schedule is quite accurate, especially when the partition is balanced and there is enough space to schedule all instructions in the cluster as specified in 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.

```
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).
```

Figure 4. Pseudocode for the register pressure estimation.

3.2. Scheduling the Instructions

Once a partition has been computed, each instruction is scheduled in the assigned cluster. This scheduling is a bi-directional scheme borrowed from the approach presented in URACAM [6], which was shown to be very effective in terms of exploiting parallelism and reducing register pressure. The URACAM scheduler gives priority to nodes according to their criticality and tries to avoid extending lifetimes. URACAM also tries to maintain balance across all critical resources and provides on-the-fly spill code generation.

Whenever an instruction cannot be scheduled in the cluster assigned by the partition, the other clusters are tried. We rely on URACAM heuristics to select the best cluster. If the instruction cannot be scheduled in any cluster, then the II is increased and the partition is refined.

3.3. Refine the Partition

Once the II is increased, extra slots may remain in clusters. The refining step starts from the previous partition and tries to utilize these remaining slots. First of all, the subgraphs that correspond to the nodes in each cluster are coarsened following the same algorithm as the one described in Section 3.1.1. Then, the partition is improved by using the enhancing heuristics described in Section 3.1.4. Note that we do not have to consider balancing memory and FUs during this step since this was done previously.

4. Evaluation

Our algorithms have been implemented using the ICTINEO compiler framework [2] and evaluated for the

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
Arch VI	4	32	2

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.

SPECfp95 benchmarks. The performance figures shown in this section refer to the modulo scheduling of the innermost loops. The 678 loops studied represent 95% of the total execution time of the programs. We study six different architectures, which are defined in Table 1. All architectures assume a single inter-cluster 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 two different scheduling algorithms:

- **baseline** - A graph partition-based approach [1], that produces a partition without the aid of a pseudo-scheduler, and that is focused on reducing the number of register bus communications.
- **PSP** - A graph partition-based approach that utilizes pseudo-scheduler directed partitioning algorithms, as described in section 3.

Our performance metric is 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.

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

Table 3. Operation latencies.

We present results for 32 and 64 total registers. Recall that the number of registers per cluster will be the number of total registers divided by the number of clusters. The smaller this ratio becomes, the more important careful management of register pressure will be.

The key feature of the new algorithm is that it considers all resources when computing the partition. Thus, the largest benefit should be obtained for the most constrained configurations. For instance, the baseline algorithm used in this paper did not take into consideration register pressure, so we should expect larger benefits when register resources are limited. When using 64 registers, the most constrained resource is usually the register bus. Since the baseline algorithm focuses on reducing register bus contention, the advantages of pseudo-scheduler directed approach should be moderate.

In Figures 5, 6, and 7, we present IPC numbers for configurations with 64 registers (ArchI, ArchII, and ArchIII, respectively). As expected, we slightly outperform the baseline for ArchI and ArchII. We obtain higher IPC for all programs except swim for ArchI and ArchII, and fpppp for ArchII. On average, the new algorithm is better. For ArchIII we obtain a bigger benefit, around 10%. To explain this gain, it is necessary to point out an interesting observation: an effective way of reducing the number of communications is to keep instructions that share a common set of operands in a single cluster. This produces two negative side-effects. First, register pressure will be increased since a large number of values will be live in a single cluster. Second, functional unit pressure will increase, which in turn extends the length of the schedule as well as register lifetimes.

By utilizing our pseudo-scheduler, we can anticipate these side-effects, keeping them balanced as we arrive at a refined partition. Since the baseline scheme does not take into account register pressure when computing the partition, if register bus pressure is high (as it is in the case of ArchIII), the baseline algorithm tends to concentrate a lot of instructions in a single cluster. This, in turn, increases register pressure. This is the reason why our pseudo-scheduler guided approach obtains a significant improvement for ArchIII.

When moving from 64 to 32 registers, careful management of register pressure and functional unit assignment becomes critical. Our pseudo-scheduler should be a lot more effective for these constrained architectures. In Figure 8 (2 clusters with 32 registers, and a 1-cycle register bus latency) we outperform the baseline in all programs except swim. In Figures 9 and 10 we show the results for 4 clusters with 32 registers (ArchV and ArchVI). In these architectures we outperform the baseline algorithm for all the programs. As we expected, the gain for these configurations is larger than

the gains obtained for architectures with 64 registers.

By utilizing a pseudo-schedule to obtain precise information about the most constrained resources, our proposed algorithm can significantly improve performance. The more constrained an architecture, the larger the benefits of using a pseudo-schedule should be. On the other hand, the performance of these architectures is far from the performance obtained for a unified architecture possessing the same resources. For example, on average we obtain a 38.5% improvement for ArchVI, increasing IPC from 1.87 to 2.59. For a unified architecture comparable to ArchVI, an IPC of 4.2 was obtained (as reported in [1]).

One down side of our approach is that compile time increases (by an order of 10x when compared with the baseline). However, since the compile time for the baseline algorithm was shown to be quite short, and VLIW architectures depend upon aggressive compilation to obtain performance, this increase in compile time is not a major concern.

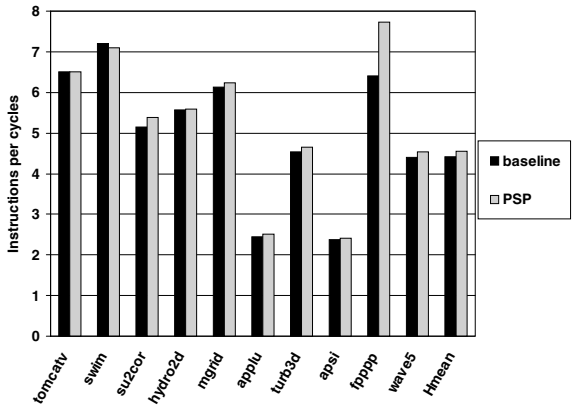

Figure 5. IPC numbers for an architecture with 2 clusters, 64 registers, 1 register bus with a 1 cycle register bus latency (Arch I)

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 an attempt to find near-optimal schedules. The objectives of past heuristic-based approaches have had different goals: increasing throughput [20, 32], minimizing register pressure [9, 8], reducing the effect of the cache misses, or improving several objectives simultaneously [8, 19, 26, 34]. All of these studies focused on modulo scheduling algorithms targeting unified (i.e., non-partitioned) architectures. A comparison of some of these techniques can be found in [5].

There are several works related to acyclic code scheduling for clustered architectures [3, 7, 10, 21, 30]. The most closely related work to our ideas include the work of Kailas, Ebcioglu and Agrawala [22]. They proposed an approach to cluster assignment, instruction scheduling and register allocation in a single compilation phase, all based on a list scheduling scheme. The approach taken in [22] differs from the approach presented in this paper in that they target instruction scheduling for acyclic code and use different cluster assignment heuristics.

A number of modulo scheduling approaches, targeting clustered VLIW architectures, have been recently proposed:

- Nystrom and Eichenberger [29] investigated cluster assignment for modulo scheduling, mainly focusing on minimizing execution overhead due to inter-cluster communication with a two-step approach:

    1. first partitioning the dependence graph of the loop body (assigning each operation to a cluster), and
    2. then scheduling the operations following the graph partition.

- Fernandes et al. [12] 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 [36] 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 [35], and further extended by Gibert et al. [16] to consider an interleaved cache.

- Codina et al. [6] presented *URACAM*, which is a framework to deal with instruction scheduling, cluster assignment and register allocation in a single phase, including a unique approach to insert spill code on-the-fly and provides effective mechanisms to deal with communications, register and memory pressure at the same time.

- Zalamea et al. [38] also proposed a technique to cluster assignment, instruction scheduling and register allocation based on an iterative scheme [32] with some heuristics to deal with spill code on-the-fly [37].

- 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

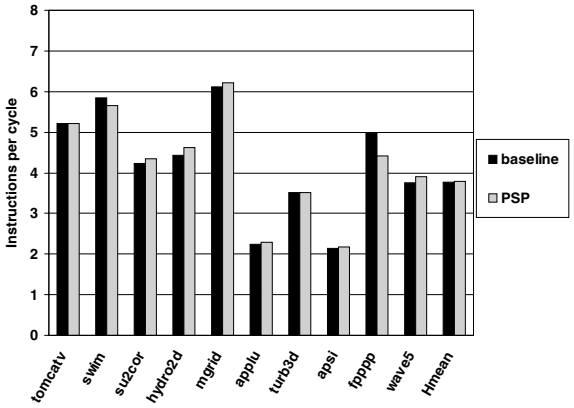

Figure 6. IPC numbers for an architecture with 4 clusters, 64 registers, 1 register bus with a 1 cycle register bus latency (Arch II)

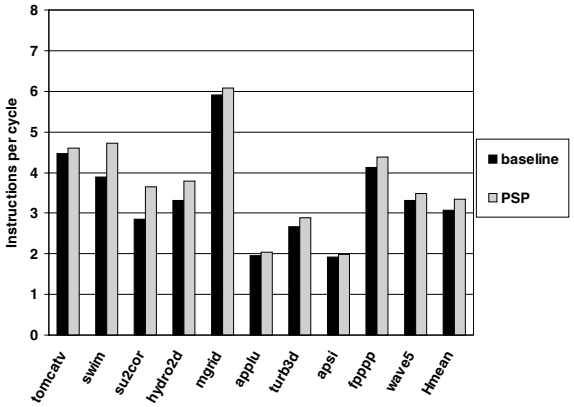

Figure 7. IPC numbers for an architecture with 4 clusters, 64 registers, 1 register bus with a 2 cycle register bus latency (Arch III)

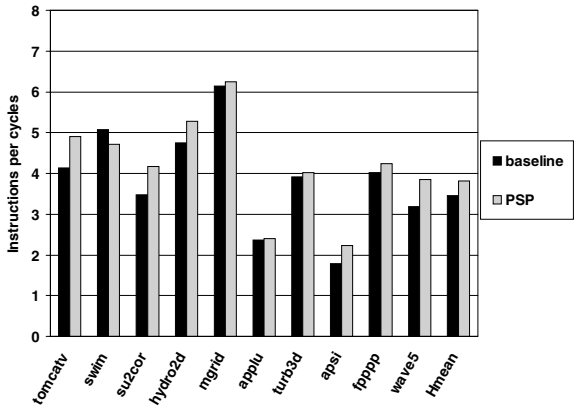

Figure 8. IPC numbers for an architecture with 2 clusters, 32 registers, 1 register bus with a 1 cycle register bus latency (Arch IV)

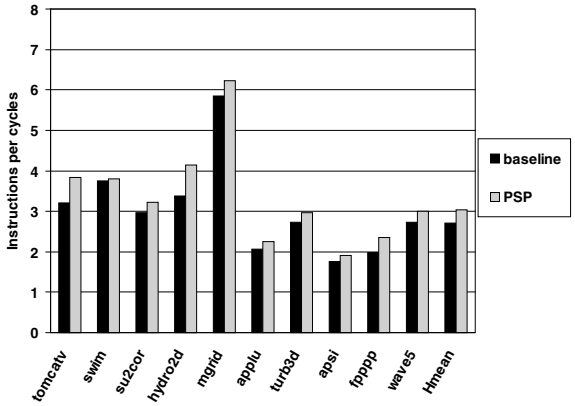

Figure 9. IPC numbers for an architecture with 4 clusters, 32 registers, 1 register bus with a 1 cycle register bus latency (Arch V)

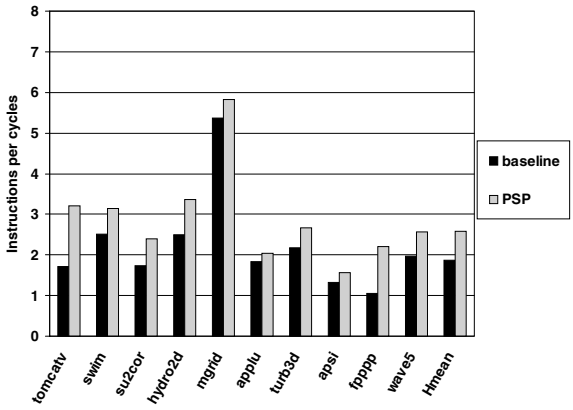

Figure 10. IPC numbers for an architecture with 4 clusters, 32 registers, 1 register bus with a 2 cycle register bus latency (Arch VI)

for a technique that combines cluster assignment, instruction scheduling and register allocation in single phase [6] 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 VLIW processors. The main novelty in our work is the use of a pseudo-scheduler that guides the graph partitioning process.

We have compared the proposed scheme with a state-of-the-art approach that is also based on graph partitioning algorithms. Results show that if we exploit an estimate of the load placed on critical system resources during partitioning decisions, we can produce better schedules than previous approaches.

In future work we plan to consider the effects of memory latency during graph partitioning.

7. 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] A. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of The AP120B/FPS-164 Family. *Computer*, 14(9):18–27, 1981.

[5] J. M. Codina, J. Llosa, and A. González. A Comparative Study of Modulo Scheduling Techniques. In *Procs. of the 16th Int. Conf. on Supercomputing*, pages 97–106, 2002.

[6] 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.

[7] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers. Technical Report HP-98-13, HP Labs Technical Report, Jan 1998.

[8] 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 Microarchitecture*, pages 338–349, 1995.

[9] A. Eichenberger, E. Davidson, and S. Abraham. Optimum Module Schedules for Minimum Register Requirements. In *Proc. of Supercomputing '95*, 1995.

[10] J. Ellis. *Bulldog: A Compiler for VLIW Architecture*. MIT Press, Cambridge, MA, 1986.

[11] 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.

[12] 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.

[13] C. Fiduccia and R. Mattheyes. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. of 19th Design Automation Conference*, pages 175–181, 1982.

[14] J. Fridman and Z. Greenfield. The TigerSharc DSP Architecture. *IEEE Micro*, pages 66–76, Jan-Feb 2000.

[15] H. Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*. PhD thesis, Stanford University, 1973.

[16] E. Gibert, J. Sánchez, and A. González. An Interleaved Cache Clustered VLIW Processor. In *Proc. of the 16th Int. Conf. on Supercomputing*, pages 210–219, 2002.

[17] B. Hendrickson and R. W. Leland. A Multi-Level Algorithm For Partitioning Graphs. In *Supercomputing*, 1995.

[18] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proc. of the IEEE*, pages 490–504, April 2001.

[19] R. Huff. Lifetime-Sensitive Modulo Scheduling. In *Proc. of the Int. Conf. on Programming Languages, Design and Implementation*, pages 318–328, 1993.

[20] 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.

[21] 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.

[22] K. Kailas, K. Ebcioglu, and A. Agrawala. CARS: A New Code Generation Framework for Clustered ILP Processors. In *Proc. of the 7th Int. Symposium on High Performance Computer Architecture*, pages 133–143, 2001.

[23] B. Kernighan and S. Lin. An Effective Heuristic Procedure for Partitioning Graphs. *Bell Syst. Tech. Journal*, pages 291–307, 1970.

[24] 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.

[25] J. Llosa, E. Ayguadé, A. González, M. Valero, and J. Eckhardt. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Transactions on Computers*, 50(3):234–249, 2001.

[26] J. Llosa, M. Valero, E. Ayguadé, and A. González. Modulo Scheduling with Reduced Register Pressure. *IEEE Transactions on Computers*, 47(6):625–638, 1998.

[27] MAP1000. MAP1000 unfolds at Equator. *Microprocessor Report*, 12(16), Dec 1998.

[28] Maximum Weighted Matching in General Graphs, *Algorithmic Solutions Software GmbH*, `http://www.algorithmic-solutions.com`, March 2001.

[29] E. Nystrom and A. E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *Proc. of the 31st Int. Symposium on Microarchitecture*, 1998.

[30] 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.

[31] G. Pechanek and S. Vassiliadis. The ManArray Embedded Processor Architecture. In *Proc. of 26th Euromicro Conference*, pages 348–355, Sept 2000.

[32] B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of 27th Int. Symp. on Microarchitecture*, pages 67–74, Nov 1994.

[33] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proc. of the 14th Annual Microprogramming Workshop*, pages 183–198, October 1981.

[34] J. Sánchez and A. González. Cache Sensitive Modulo Scheduling. In *Proc. of 30th Int. Symp. on Microarchitecture*, pages 338–348, Dec 1997.

[35] 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.

[36] 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.

[37] 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.

[38] 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.