Distributed modulo scheduling

Distributed Modulo Scheduling

Marcio Merino Fernandes
University of Edinburgh, UK
Department of Computer Science
mmf@dcs.ed.ac.uk

Josep Llosa
Universitat Politècnica de Catalunya, Spain
Department d'Arquitectura de Computadors
josepll@ac.upc.es

Nigel Topham
University of Edinburgh, UK
Department of Computer Science
npt@dcs.ed.ac.uk

Abstract

Wide-issue ILP machines can be built using the VLIW approach as many of the hardware complexities found in superscalar processors can be transferred to the compiler. However, the scalability of VLIW architectures is still constrained by the size and number of ports of the register file required by a large number of functional units. Organizations composed by clusters of a few functional units and small private register files have been proposed to deal with this problem, an approach highly dependent on scheduling and partitioning strategies. This paper presents DMS, an algorithm that integrates modulo scheduling and code partitioning in a single procedure. Experimental results have shown the algorithm is effective for configurations up to 8 clusters, or even more when targeting vectorizable loops. [1]

Keywords: ILP, VLIW, Clustering, Software Pipelining

1. Introduction

Current microprocessor technology relies on two basic approaches to improve performance. One is to increase clock rates, resulting in faster execution of machine operations. The other is *instruction-level parallelism (ILP)*, a set of hardware and software techniques that allows parallel execution of machine operations. ILP can be exploited by VLIW architectures [8, 16]. In this case all data dependence analyses and scheduling of operations are performed at compile time, which simplifies the hardware and allows the inclusion of a large number of functional units in a single chip.

[1]Research work partially supported by Capes (Brazil)

Loop structures usually found in DSP or numeric applications can take advantage of the available processing power of a wide-issue machine. In many cases they account for the largest share of the total execution time of a program. Several loop optimizations have been developed targeting ILP machines. One of them is *software pipelining* [2], a scheduling technique that allows the initiation of successive loop iterations before prior ones have completed. Modulo scheduling is a class of software pipelining algorithms that produces a basic schedule for a single iteration [15]. The basic schedule is structured in order to preserve data dependencies and avoid machine resource conflicts if it is issued every *Initiation Interval (II)* cycles [14].

The drawback of these techniques is that they increase *register requirements* [10]. The number of storage positions alone can be a problem in the design of a register file (RF). Furthermore, the number of ports required by a VLIW machine may compromise the RF access time, causing a negative impact on the machine cycle time [4]. Hence, wide-issue unclustered VLIW architectures may not deliver the expected performance, which has motivated us to develop a clustered VLIW architecture [7]. However, the effectiveness of such an organization also depends on the code partitioning strategy, as data dependent operations must communicate results between them. We have developed a scheme to produce software pipelined code for a clustered VLIW machine aiming to achieve performance levels similar to an unclustered machine without communication constraints. It is called **Distributed Modulo Scheduling (DMS)**, *integrating* in a single phase both scheduling and partitioning of operations. The remaining of this paper includes an overview of the architecture model targeted by DMS, presents the algorithm, and shows some experimental results along with related conclusions.

2. A Clustered VLIW Architecture

The structure of the clustered VLIW architecture targeted by DMS is shown in figure 1. It comprises a collection of clusters connected in a *bi-directional ring* topology. In this paper we focus exclusively on the performance of the VLIW compute-engine, as it should determine the performance of execution of the target applications for this kind of architecture.

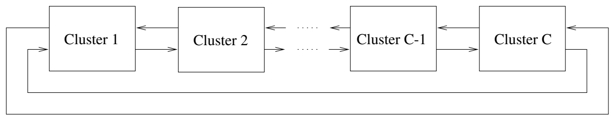

Figure 1. Clustered VLIW architecture

Each cluster contains a set of functional units (FUs) capable of executing a statically compiled loop schedule. They connect to a *Local Register File (LRF)*. We have shown in [5] that loop variant lifetimes produced by a modulo scheduled loop can be allocated to a *queue register file*, resulting in some advantages over a conventional RF. Hence, all *intra-cluster* communication takes place via the LRF, while *inter-cluster* communication takes place via one of the *Communication Queue Register Files (CQRFs)*. A CQRF is a queue register file located between two adjacent clusters, providing read-only access to one of them, and write-only access to the other. Sending a value from one cluster to another requires only a pair of write/read operations to the appropriate CQRF. Thus, no explicit instruction is necessary for near-neighbour communication. This is done by the code generator, which maps lifetimes that span a cluster boundary onto the corresponding CQRF. One of the advantages of this communication mechanism is to allow *fixed timing* in the communication process between two clusters, a desirable feature for static schedulers. Another motivation for using queues is the possibility of implementing asynchronous data transfer across clusters, which might be necessary due to clock skewing.

In spite of the distribution of functional units among clusters, the proposed architecture model still assumes a single thread of control. This will almost certainly involves data exchange among FUs located in distinct clusters. Compiling for a clustered architecture involves *code partitioning* in order to meet communication constraints. An optimal partitioning would yield in the same performance that would be otherwise achieved by an unclustered architecture. However, communication constraints may require a group of operations to be scheduled in a given cluster, which may not have enough resources for that. In this case, the only alternative is to increase the II, reducing the net execution rate.

A number of previous works have dealt with problems similar to this. The Multiflow Architecture [11] performs code partitioning and then scheduling of operations in two separate steps. The Limited Connectivity Model also performs these phases in sequence, though the other way around [1]. A two-phase approach to partitioning and modulo scheduling for a clustered architecture is proposed in [6]. The idea is to partition prior to scheduling, ensuring that no communication conflicts arise when operations are scheduled. This problem can be described as a *k*-way graph partitioning in which the II is to be minimized. Once the partitioning is completed, the scheduling can proceed, taking into account the assignment of operations to clusters. A similar scheme was also reported in [12]. Experiments with an algorithm integrating in a single phase both modulo scheduling and code partitioning was presented in [7]. Although effective for machine models with up to 5 clusters, the scheme is inappropriate for larger configurations because it cannot consider communication between indirectly-connected clusters. That algorithm originated DMS, which addresses this problem. Another algorithm combining both tasks in a single phase is UAS [13]. In that scheme cluster assignment is integrated into a list scheduler, although software pipelining is not performed.

3. DMS Algorithm Description

We have used the Iterative Modulo Scheduling (IMS) algorithm [14] as the basic structure to develop DMS, a scheme able to deal with distributed functional units and register files. As defined in [14], we assume that a *data dependence graph (DDG)* is used to represent the dependencies between operations of the innermost loop to be scheduled. A clustered machine model introduces communication constraints to the scheduling algorithm, in addition to resource and dependence constraints. We say that a **communication conflict** occurs when two operations with a *true data dependence* are scheduled in indirectly-connected clusters.

IMS has one basic strategy to find a *valid slot* to schedule a given operation *OP*, which takes into account its scheduled predecessors and resource conflicts. The later can lead to *backtracking* in order to unscheduled operations to release a slot for *OP*. Eventually, successor operations of *OP* might also be unscheduled, if a dependence conflict arises. On the other hand, the DMS algorithm has three basic strategies to schedule an operation, as seen in figure 2.

Initially DMS tries to find a valid slot to schedule *OP* in such a way that no communication conflict arises with its scheduled *predecessors* and *successors* (strategy 1). In this case a slot is considered valid to schedule *OP* only if the communicating operations in the resulting partial schedule are located in *directly* connected clusters.

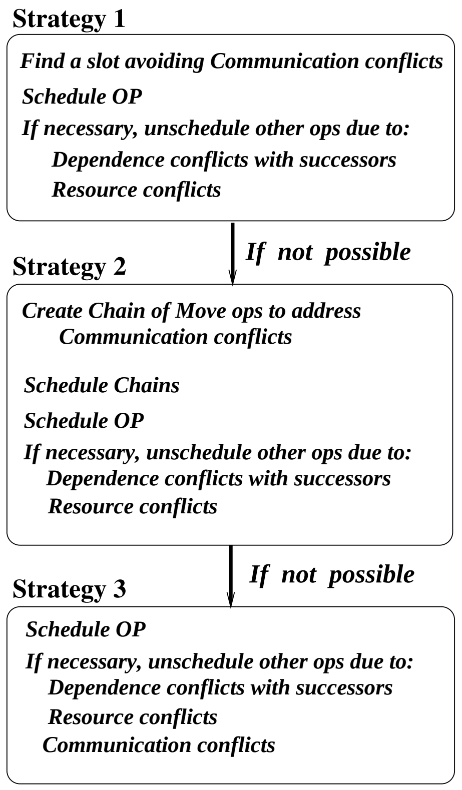
Strategy 1
Find a slot avoiding Communication conflicts
Schedule OP
If necessary, unschedule other ops due to:
Dependence conflicts with successors
Resource conflicts
If not possible
Strategy 2
Create Chain of Move ops to address Communication conflicts
Schedule Chains
Schedule OP
If necessary, unschedule other ops due to:
Dependence conflicts with successors
Resource conflicts
If not possible
Strategy 3
Schedule OP
If necessary, unschedule other ops due to:
Dependence conflicts with successors
Resource conflicts
Communication conflicts

Figure 2. Overview of DMS algorithm

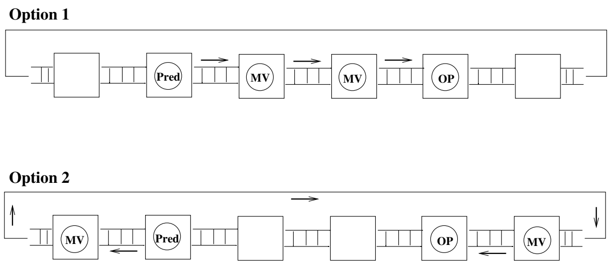

Figure 3. Options to create a chain

If that is not possible, strategy 2 is attempted. In this case DMS tries to insert **move** operations between *OP* and all of its *scheduled predecessors*, using a structure called **chain**. A chain is a string of move operations scheduled in the clusters between *OP* and one of its predecessors. This makes possible to transfer operands between a pair of producer/consumer operations located in indirectly-connected clusters. In the particular architecture model considered in this paper, a move operation simply read one value from a CQRF and write it back to another one. Thus, given a candidate cluster to schedule *OP*, and the cluster of its predecessor, there are two possibilities to create a chain, each of them following *opposite* directions (figure 3). The bi-directional ring of queues used to connect clusters allows this flexibility.

Initially any cluster can be considered to schedule *OP*. More than one chain might be necessary to schedule *OP* in a given cluster because multiple predecessors may be already scheduled. However, these chains can be built only if there are enough machine resources to schedule *all* move operations in the respective clusters. As above discussed, more than one option to schedule a chain might exist to address a given communication conflict. In this case, the selected option is the one that maximizes the number of free slots left available to schedule move operations in *any* cluster. If two or more possibilities are equivalent regarding this criteria, the smallest number of move operations defines the choice. These conditions determine the cluster in which *OP* will be scheduled.

Once a valid set of chains is chosen, it can be scheduled straightforward as the availability of machine resources has already been verified. The first step involves updating the DDG to include the new move operations and related data dependencies. Then move operations are sequentially scheduled, starting from the first one after the original producer operation. This ordering must be enforced to determine the correct scheduling time of each of them.

If resource conflicts prevent the use of chains to overcome communication conflicts, *OP* is scheduled in a arbitrarily chosen cluster using a process similar to the one employed by IMS. The only difference is that the backtracking process must also unschedule some operations due to communication conflicts (strategy 3).

Special attention must be paid in the implementation of the backtracking procedures. It might happen that an operation ejected from the partial schedule is part of a chain. In this case it may also be necessary to unschedule other operations and update the DDG in order to prevent communication conflicts with the remaining scheduled operations. Distinct actions must be taken when the unscheduled operation is the original producer, a move operation, or the original consumer, respectively.

It is expected that the additional constraints used by DMS may increase the backtracking frequency. However, we have found through experimental analysis that the overhead on the II due to partitioning is tolerable in most of the cases (section 4). Those results suggest that on average the backtracking frequency of IMS and DMS are of the same order. When the backtracking frequency increases it is usually due to insufficient number of slots to schedule the required move operations, rather than a lengthy search across the space of solutions.

Although DMS has been specially developed for the architecture model described in section 2, we believe it could also be used with other clustered VLIW architectures. We understand that other candidate architectures should possess three basic characteristics in order to use DMS efficiently:

- Directly-connected clusters should communicate through a mechanism able to ensure fixed timing constraints, known at compile time.
- The number of possible paths to create a chain should be small, in order to avoid searching through an excessive number of options.
- Some sort of DDG transformation should be made in order to limit the number of immediate data dependent successors of an operation.

The CQRF used in the architecture model presented in section 2 allows a value to be read only once from any of its FIFO queues. Thus, prior to modulo scheduling, all multiple-use lifetimes are transformed into single-use lifetimes using *copy* operations, as reported in [7]. This transformation has also the effect of limiting the number of immediate successors of any operation to 2, which simplifies the code partitioning among clusters with limited connectivity. Multiple-use lifetimes would concentrate the number of move operations around the original producer, possibly requiring more scheduling slots than available within the sough II.

4. Experimental Results

We have used an experimental framework to perform modulo scheduling and register allocation of loops for several architecture configurations, some of them presented in this section. Two architecture models have been considered: unclustered and clustered, which were scheduled using IMS and DMS, respectively. The machine configurations range from 1 to 10 clusters, each of them having 3 functional units: 1 L/S, 1 ADD, and 1 MUL. In addition, each cluster has also a Copy FU to perform copy and move operations. However, these functional units and operations are not considered to estimate performance figures, as they do not perform any useful computation, All eligible innermost loops from the Perfect Club Benchmark have been used, a total of 1258 loops suitable for software pipelining. The original body of many of those loops do not present enough parallelism to saturate the FUs of wide-issue machines. Hence, *loop unrolling* was performed to provide additional operations to the scheduler whenever necessary [9].

As already discussed, a good scheduling/partitioning algorithm should minimize an eventual increase of the II in relation to the value otherwise achieved for the corresponding unclustered machine. The data in figure 4 shows the fraction of loops presenting any increase in the II due to DMS partitioning. Overheads for machines with 2 and 3 clusters are only due to the introduction of copy operations in the DDG, as no communication conflicts occurs in these cases. Over 80% of the loops do not present any overhead for machine models up to 8 clusters (24 FUs). When the II increases it is mainly because the Copy FUs became the most heavily used resources, due to an excessive number of move operations. That could be improved with additional hardware support.

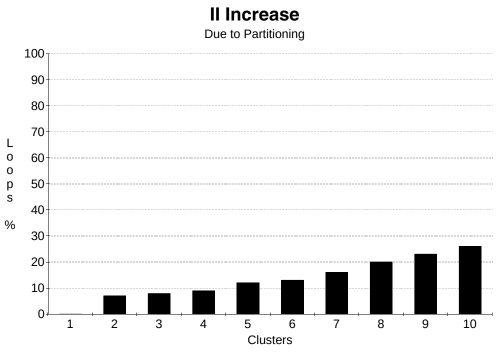

Figure 4. Overhead on II due to partitioning

Performance analyses regarding the execution of two sets of loops were done. Set 1 comprises *all* loops of the benchmark, while set 2 contains only loops without recurrences [14]. The second set was considered because those loops are highly vectorizable, having characteristics similar to the ones usually found in DSP applications [3]. Hence, they can take more advantage of additional machine resources.

The data in figure 5 shows the total number of cycles (in relative values) required to execute the modulo scheduled loops in each machine configuration. The difference between clustered and unclustered machines shows that the partitioning process results only in small performance degradation for up to 21 FUs when the set 1 is used. However, the difference is more accentuated when wider-issue machines are used. On the other hand, very small differences are observed if only loops without recurrences are considered. Furthermore, the results suggest that DMS may be effective with these loops for even wider-issue machines.

The data in figure 6 shows the number of instructions issued per cycle (IPC). It was measured taking into account the iteration counter, including operations from the kernel code, prologue, and epilogue phases. If all loops are considered, the IPC value improves for machines up to 21 FUs (7 clusters), however it levels beyond that point. Loops without recurrences allow improvements for the whole range of machine models, which confirms that they are better suited to exploit ILP in this kind of architecture.

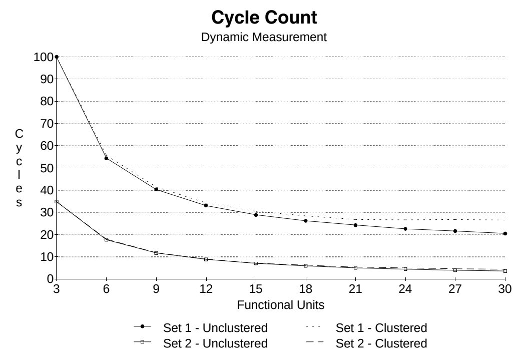

Figure 5. Execution time

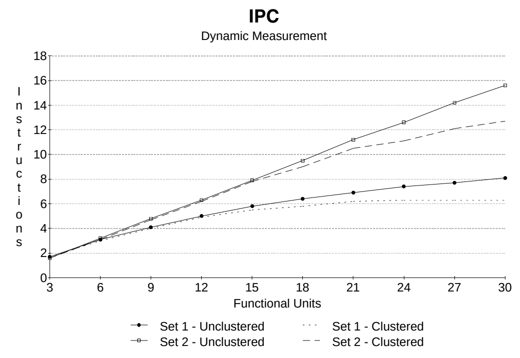

Figure 6. IPC-Instruction per cycle

5. Conclusions

The proposed DMS algorithm is effective for machine configurations up to 8 clusters, resulting in low overhead due to partitioning. A larger overhead was observed for wider-issue machine, although that could be minimized by using additional FUs to schedule move operations. In most of the cases, the use of a few move operations is enough to avoid dead-end states due to communication conflicts. DMS can produce efficient software pipelined code for clustered VLIW machines comprising a number of clusters not previously considered in other works, to the best of our knowledge. Hence, it can significantly extend the potential for ILP exploitation in this kind of architecture, which may be particularly suitable for DSP and numeric applications.

References

[1] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of trade-offs. In *Proceedings of the MICRO-25 - The 25th Annual International Symposium on Microarchitecture*, 1992.

[2] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9), 1981.

[3] P. Faraboschi, G. Desoli, and J. Fisher. The latest word in digital and media processing. *IEEE Signal Processing Magazine*, March 1998.

[4] K. Farkas, N. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. Technical Report 95/10, Digital Western Research Laboratry, 1995.

[5] M. Fernandes, J. Llosa, and N. Topham. Allocating lifetimes to queues in software pipelined architectures. In *EURO-PAR'97, Third International Euro-Par Conference*, Passau, Germany, 1997.

[6] M. Fernandes, J. Llosa, and N. Topham. Extending a VLIW architecture model. Technical Report ECS-CSG-34-97, University of Edinburgh, Department of Computer Science, 1997.

[7] M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered VLIW architectures. In *IPPS'98, 12th International Parallel Processing Symposium*, Orlando, USA, 1998.

[8] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

[9] D. Lavery and W. Hwu. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, 1995.

[10] J. Llosa, M. Valero, and Ayguadé. Quantitative evaluation of register pressure on software pipelined loops. *International Journal of Parallel Programming*, 26(2):121–142, 1998.

[11] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, and R. Nix. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, July 1993.

[12] E. Nystrom and A. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the MICRO-31 - The 31th Annual International Symposium on Microarchitecture*, 1998.

[13] E. Ozer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the MICRO-31 - The 31th Annual International Symposium on Microarchitecture*, 1998.

[14] B. Rau. Iterative modulo scheduling. *The International Journal of Parallel Programming*, February 1996.

[15] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Workshop on Microprogramming*, 1981.

[16] R. Rau and J. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, July 1993.