Partitioned Schedules for Clustered VLIW Architectures

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

This paper presents results on a new approach to partitioning a modulo-scheduled loop for distributed execution on parallel clusters of functional units organized as a VLIW machine. A distinctive characteristic of this architecture is the use of register files organized by means of queues, which results in a number of advantages over conventional schemes, but also requires the development of specific compiling and hardware features. We have investigated a scheme based on copy operations to deal with data values to be consumed more than once during loop execution. Experiments with loop unrolling were also performed in order to optimize both loop execution and the use of machine resources. A partitioning algorithm has been implemented to perform some experiments with the clustered architecture model, an organization widely accepted as being essential for very wide issue machines. [1]

Keywords: Parallel Processing, VLIW, Software Pipelining, Modulo Scheduling

1. Introduction

Very Long Instruction Word (VLIW) architectures [9] exploit *instruction level parallelism* (ILP) without the need for run-time data dependence analysis as this is performed at compile time, resulting in a simpler hardware organization when compared to a superscalar processor. This allows the inclusion of a larger number of functional units (FU) into a single chip, increasing the possibilities for parallelism. Sophisticated compiling techniques have been developed in order to identify parallelism and schedule operations for ILP machines, such as *software pipelining* [4], a technique that allows the initiation of successive loop iterations before prior ones have completed. One class of software pipelining algorithms is *modulo scheduling* [16], an efficient scheme to optimize the use of machines with complex resource patterns.

The ideal VLIW machine has a number of concurrent FUs, connected to a register file able to perform two reads and one write *operation* (op) per functional unit in each cycle [3], thus requiring complex hardware organizations with a possible increase in access time. Furthermore, most software pipelining schemes assume that arithmetic operations are all register-register operations and data is transferred between registers and memory using load and store instructions. The time span from the reservation of a register to hold a value up to the last cycle before the value is used is called a *lifetime*. In a software pipelining scheme multiple iterations can be initiated before previous ones have completed, which means that lifetimes from the same operation may coexist, thus requiring distinct storage locations and increasing register pressure [14]. The use of a *conventional register file* (RF) is not a straightforward solution, even for a modest number of functional units, and is unacceptable in terms of scalability [8]. Early designs proposed alternative register file organizations to deal with the problem, among them the Polycyclic architecture [16] and the Cydra machine [17]. Alternative architecture models comprising clusters of FUs and small private register files have been proposed [3, 12]. Although this approach indeed reduces the hardware complexity of individual register files it imposes further constraints on both the scheduler and the register allocator, which may result in performance degradation if not properly handled.

[1] This work has been supported in part by research grants from Capes (Brazil), British Council and Ministry of Education of Spain under Acciones Integradas grants No. 1016 and 202b

Our approach to the problem consists in the design of a scalable VLIW architecture comprising clusters of functional units and private register files implemented as queue structures (QRF), which in turn may also be used as a mechanism for inter-cluster communication [6]. Register files organized by means of queues are believed to be less complex than conventional organizations [1, 10, 11]. However this hardware simplification imposes new constraints on the register allocator, requiring new techniques to efficiently exploit such organization. We have taken advantage of the regular pattern of production and consumption of lifetimes resulting from modulo schedules to deduce and prove a *Compatibility Test* [8] to decide if values produced by distinct operations are *Q-Compatible*, which means they can be stored in the same queue. The basic idea is that in a modulo schedule two or more lifetimes can share a common storage queue as long as their production order exactly matches their consumption order, which can be checked by applying the following theorem:

Theorem 1.1 *Let II be the Initiation Interval in cycles between two consecutive loop iterations. Two computations* **a** *and* **b***, with start-times S_a and S_b, and lifetimes L_a and L_b such that $L_a \geq L_b$, are Q-Compatible* if and only if $L_a - L_b < (S_b - S_a) \ mod \ II$.

The following sections of this paper present and discuss some of our latest findings towards the design of a clustered VLIW architecture model using queue register files. We believe this approach compares favourably against conventional ones on a number of aspects, including the required silicon area, register name space, register allocation, code generation and facilities to implement a scalable clustered machine [7].

2. Dealing with Simultaneous Writes

The architecture model being developed assumes that values produced by an operation are stored in a register file by means of a write op, to eventually be consumed by another operation(s) by means of a read op. As seen in the *data dependence graph* (ddg) in Fig. 1(a), a value produced by a given operation my be consumed by more than one operation. If a conventional register file is used just one write op is necessary, no matter how many times this value will be read (Fig. 1(b)). However in our QRF model a value can be read from a queue only once, being destroyed afterwards. This implies that if a value is consumed by more than one operation it must be stored in distinct queues (Fig. 1(c)), requiring simultaneous write operations. Among other problems this would complicate the instruction format and the access to queues.

To deal with the problem we propose the introduction of a copy operation, which should be executed by a dedicated

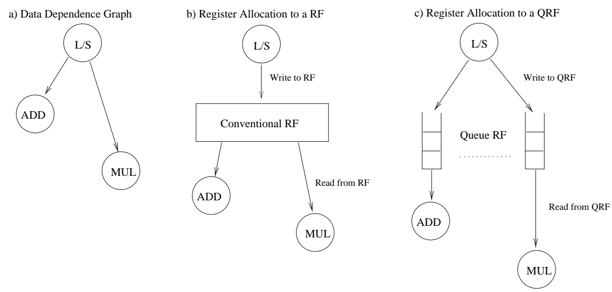

Figure 1. Register Allocation Data Flow

functional unit capable of reading one value from a queue and writing it back to two other queues, as shown in Fig. 2. In terms of hardware cost it requires only an extra FU and the corresponding register file ports, which should be simple to implement.

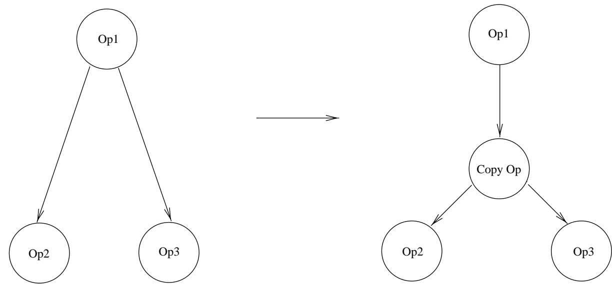

Figure 2. Including a Copy Operation

Extra delays may increase the loop execution time due to alterations in the ddg to include copy operations, so we performed some experiments and compared the results against the ones obtained for a basic configuration not using copy operations [7]. The experimental framework consists of machine models of 4, 6 and 12 FUs, a total of 1258 innermost loops from the Perfect Club Benchmark [2], a version of Rau's *Iterative Modulo Scheduling* (IMS) [15] algorithm, and register allocators for both, conventional and queue register files, as described in detail in [7].

We found that using copy operations does not increase significantly the number of queues required to schedule a given fraction of the benchmark loops, specially for loops requiring 16 or 32 queues. It should be noticed that copy operations do not change the machine configuration required to schedule most of the loops of the benchmark, which consist of 32 queues, as seen in Fig. 3.

In a modulo scheduled loop the code execution at full performance occurs at the *kernel stage*, which accounts for the largest share of the total execution time. Our framework was able to schedule around 95% of the loops with

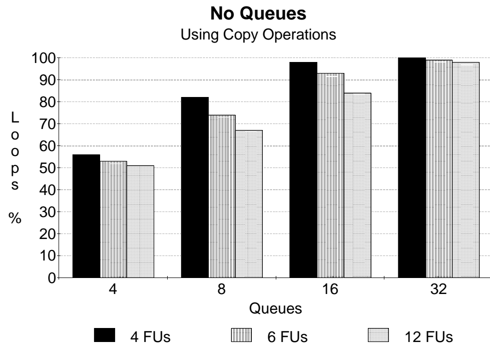

Figure 3. Number of Queues

the same II after the insertion of copy operations, which means that there is no performance degradation in the execution of the kernel for most of the cases. The remaining loops required a number of slots to allocate copy operations that could not be found within the original II, requiring an increase in its value (tolerable in most of the cases). The *stage count* is related to the number of loop iterations simultaneously in execution. A higher stage count value results in longer *prologue* and *epilogue* phases, the less efficient stages surrounding the kernel execution. We found that inserting copy operations in the ddg does not change the value of the stage count for most of the loops [7]. To summarize, the use of copy operations allows us to solve a particular problem arisen by the use of a QRF, at a cost of extra FUs and register file ports, and a small increase in execution time for 5% of loops. An interesting side effect observed is a small reduction in the required number of queues and queue positions for the most demanding loops.

3. Increasing Parallelism Exploitation

One of the main objectives of this research work is to design a scalable VLIW machine. The original code of loop bodies does not always present enough operations to take full advantage of a highly parallel machine, requiring some actions in order to avoid sub-utilization [13]. Loop unrolling [5] is a well known compiler optimization that replicates the body of a loop a given number of times, called the *unroll factor*, resulting in a larger number of available operations for parallel execution. However, unrolling can also generate side effects that may compromise the benefits achieved. In this work we are particularly interested in a possible increase in register pressure.

We performed some experiments to measure the effectiveness of using loop unrolling with this architecture model, using the experimental framework described in [7]. A parameter called $II_{speedup}$ was used to compare the execution time of the kernel code of two distinct schedules of the same loop, one of them using loop unrolling:

$$II_{speedup} = \frac{II_{original\ loop}}{II_{unrolled\ loop}} \tag{1}$$

We found that a considerable fraction of the loops achieved $II_{speedup} > 1$ when loop unrolling was applied, as seen in Fig. 4. This is very significant considering that no extra FU was used. Unrolling does not alter the stage count for most of the loops, and when changes occur the stage count often decreases. This helps to improve the overall performance.

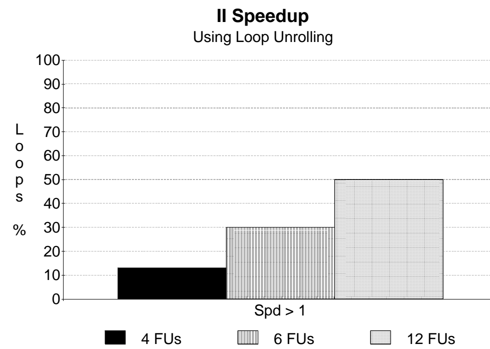

Figure 4. Initiation Interval Speedup

As expected, loop unrolling produces a moderate increase in the required number of queues, although 32 queues are still enough to schedule over 90% of the loops for any of the machine configurations considered [7]. An increase in the number of queues is less likely for the large loops because in general they do not require unrolling to exploit efficiently the machine resources.

4. Clustered VLIW Architecture

A wide issue VLIW machine organized in a single cluster would require a highly multiported register file, which needs a disproportionate chip area. The number of registers itself is a complex enough problem, however the number of access ports may be even more problematic. To illustrate that a 12 FUs machine requiring 2 read and 1 write ports for each FU would demand a 36 port register file, an unrealistic design according to current standards. Our approach to deal with the problem is to subdivide the machine into clusters, each of them comprising a few FUs and a small register file. The design and implementation

of individual clusters should not be an issue in such model, however the assignment of operations to clusters must be properly handled to conform with the inter-cluster communication topology and to minimize the II.

In the first set of experiments we defined a cluster configuration comprising 3 FUs, which are 1 L/S (Load/Store), 1 ADD (Adder) and 1 MUL (Multiplier), and also an extra FU to support copy operations as shown in Fig. 5(a). All of them connect to a private QRF. We assume that clusters are interconnected by a *bidirectional communication ring*, implemented by means of queue structures (Fig. 5(b)). These communication queues are used to allocate registers as if they were a cluster private QRF, but with the difference that a value written by a FU from a given cluster will be read by a FU belonging to another one.

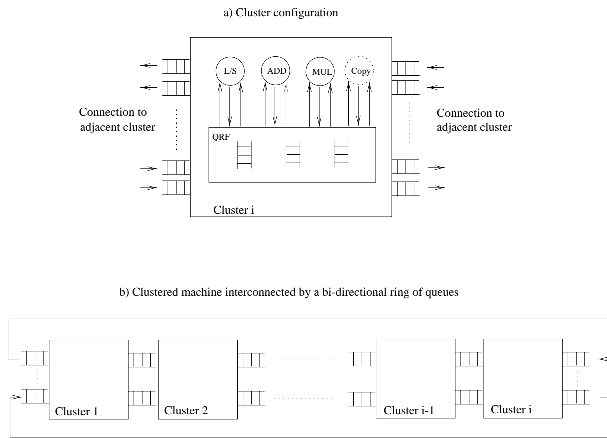

Figure 5. Clustered Machine Organization

The partitioning process is carried out by adding some heuristics to the IMS algorithm in order to avoid communication conflicts. These conflicts may arise because we do not as yet consider the introduction of operations to transfer a value between indirectly connected clusters. This limitation can prevent an operation from being scheduled in any of the clusters, leading to a *backtracking* process to unschedule conflicting operations. Backtracking may result in a larger II value that reduces the performance that would be achieved if a single cluster machine had been used instead. Accordingly, one of the aims of these first experiments is to measure how effective this partitioning algorithm is at distributing operations among clusters for the same II that would be required using a single cluster machine. We performed experiments for machine configurations of 12, 15, and 18 FUs (4, 5, and 6 clusters respect.) plus the required FUs to support copy operations. Loop unrolling was performed in all the experiments.

The data presented in Figure 6 show the fraction of loops scheduled for a clustered machine exhibiting the same II as the schedules for the corresponding single cluster machine. There is no increase in the II value for 95% of the loops when a 4 cluster machine (12 FUs) is used, and when it occurs is typically of one cycle only. When a 5 cluster machine (15 FUs) is used it is possible to schedule 84% of loops with the same II, decreasing to 52% when a 6 cluster machine (18 FUs) is used. The results indicate that the partitioning scheme adopted tends to produce poorer results as the number of clusters increases, which is mainly due to the inability to move data values between non-adjacent clusters. We have noticed that the stage count remains the same for a large fraction of the loops, suggesting that the performance of a clustered machine should not be significantly influenced by variations in the stage count [7].

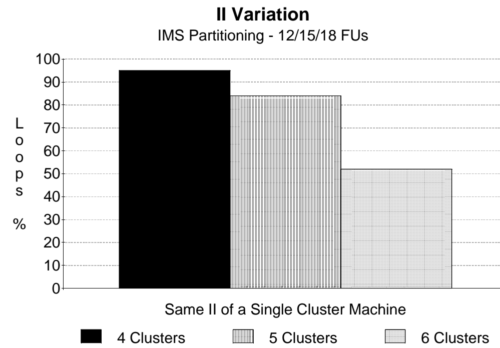

Figure 6. Initiation Interval Variation

In terms of machine resources we have found that a cluster configuration comprising 8 queues for the private QRF and another 16 queues to implement the communication ring (8 to be used in each direction) should suffice for any of the machine models analysed. Figure 7 shows the basic unit that could be used to implement a clustered machine. A small fraction of loops would require additional resources, however we believe that further improvements in partitioning are possible. Of course, in a practical system spill code will occasionally be required to deal with finite numbers of queues and queue positions.

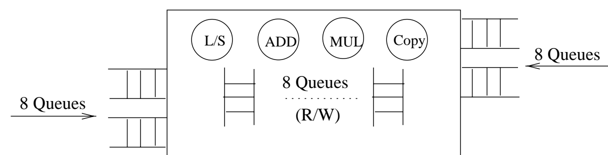

Figure 7. Basic Cluster Configuration

We performed static and dynamic analysis on the *aver-*

age number of operations issued per cycle (IPC). The static issue, IPC_{static}, accounts for the number of instructions issued in the kernel phase for one iteration. The dynamic issue, $IPC_{dynamic}$, take into account the total number of iterations performed on each loop body to estimate its execution time, also including the prologue and epilogue phases.

One of the analyses considered all loops of the benchmark, as seen in Fig. 8. Those results are compromised by the fact that several loops are simply not able to take advantage of the extra machine resources available as they are constrained by recurrence circuits in the loop body. We performed a second analysis in order to have an insight on how well this architecture model deals with programs whose execution is constrained by the number of available FUs (Fig. 9). The differences found between a single cluster and a clustered machine comprised of either 15 or 18 FUs are mainly due to the partitioning algorithm, indicating again the need of a scheme allowing move operations to transfer data between no adjacent clusters. The average instruction issue is higher for the static analysis mainly because it does not compute the less efficient prologue and epilogue phases, which is done in the dynamic analysis. For the most aggressive machine it was observed a better improvement on dynamic issue than on static issue. This happens because a few large loops, accounting for a large share of the total execution time, can take full advantage of the additional functional units, an effect that is only seen seen in dynamic analyses. This is more noticeable for clustered machines because these large loops can be scheduled without the partitioning algorithm degrading performance, which further emphasizes their influence on the overall results.

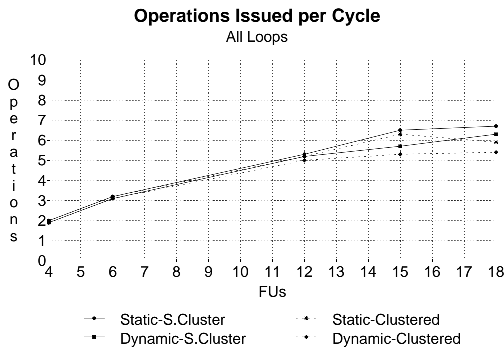

Figure 8. IPC-All Loops

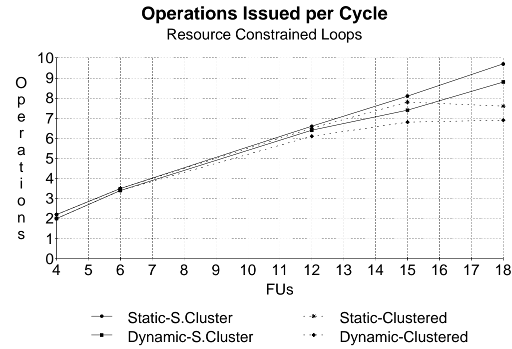

Figure 9. IPC-Resource Constrained Loops

5. Conclusion

This paper has presented results on a new partitioning strategy to be used with a clustered VLIW architecture model based on register files organized by means of queues. A scheme based on copy operations was proposed to deal with problems associated with values to be consumed more than once. Experimental results showed that it is able to solve the problem with no performance degradation in most of the cases. The use of loop unrolling resulted in dramatic improvements in loop execution and exploitation of parallelism, which was accomplished without a significant increase in the number of machine resources required. Some experiments with a clustered machine model were done, leading to a preliminary specification of the hardware configuration of each cluster, and also the communication topology among them. A partitioning algorithm has been developed, allowing us to obtain satisfactory results for machines composed of 4 and 5 clusters. However its efficiency is compromised when more clusters are used, thus requiring a more sophisticated scheme using move operations to transfer values between indirectly connected clusters. Such scheme should make possible for a clustered machine to achieve performance figures similar to the ones observed for a single cluster machine, taking full advantage of the architecture scalability. We believe that this architecture can also be used to run scalar code, however a discussion on the related techniques required is beyond the scope of this paper. In order to further improve this model we are currently working on this new partitioning technique, some strategies to deal with loop invariants, and also refining some of the hardware design specifications, including a complexity model for the queue register file.

References

[1] M. Aloqeely and C. Chen. A new technique for exploiting regularity in data path synthesis. In *EURO-DAC'94, European Design Automation Conference*, 1994.

[2] M. Berry, D. Chen, P. Koss, and D. Kuck. The perfect club benchmarks: Effective performance evaluation of supercomputers. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1988.

[3] 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.

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

[5] J. Dongarra and R.Hinds. Unrolling loops in Fortran. *Software-Practice and Experience*, pages 219–226, 1979.

[6] 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.

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

[8] M. Fernandes, J. Llosa, and N. Topham. Using queues for register file organization in VLIW architectures. Technical Report ECS-CSG-29-97, Edinburgh University, Department of Computer Science, 1997.

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

[10] S. Gerez and E. Woutersen. Assignment of storage values to sequential read-write memories. In *EURO-DAC'96, European Design Automation Conference*, 1996.

[11] A. Heubi, M. Ansorge, and F. Pellandini. A low power VLSI architecture with an application to adaptive algorithms for digital hearing aids. In *EUSIPCO-94, Seventh European Signal Processing Conference*, 1994.

[12] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, 1995.

[13] 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.

[14] J. Llosa, M. Valero, E. Ayguadé, and J. Labarta. Register requirements of pipelined loops and their effect on performance. In *2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, 1994.

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

[16] 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.

[17] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer. *Computer*, January 1989.