Allocating Lifetimes to Queues in Software Pipelined Architectures

Marcio M. Fernandes[1], Josep Llosa[2], Nigel Topham[1]

[1] Edinburgh University, UK
[2] Universitat Politècnica de Catalunya, Spain

Abstract. Software pipelining is an effective technique for increasing the throughput of loops in superscalar or VLIW machines, however it generates high register pressure, which in some cases requires the introduction of spill code into the schedule. Large multi-ported register files present significant problems in the construction of scalable VLIW systems, which has lead us to investigate architectures in which part of the register file is replaced by queues. We believe that this organization has distinct advantages in terms of hardware complexity, silicon area, instruction name space, and scalability. Queues also represent a natural mechanism for communication between clusters of functional units in a partitioned VLIW system. In this paper we present an overview of this approach, along with some experimental results suggesting it as being a feasible organization.

1 Introduction

Instruction-level parallelism (ILP) is a family of processor and compiler design techniques that speed up program execution by causing individual machine operations to execute in parallel. Decisions about which operations should be executed in a given cycle and a given functional unit can be taken either at compile time or at run time, depending on the architecture model in use. In Very Long Instruction Word (VLIW) machines the compiler provides information as to which operations are independent of one another, so the hardware knows without further checking which operations can execute concurrently.

The scheduling of operations plays a major role in achieving near optimal performance from an ILP machine. One of the scheduling schemes that can be employed is software pipelining, with the objective of initiating successive loop iterations before prior iterations had completed [2]. Modulo scheduling is a class of software pipelining algorithms in which all loop iterations have the same schedule of operations [9]. Most software pipelining schemes assume an architectural model in which arithmetic operations are all register-register operations

[0] This work has been supported by research grants from Capes (Brazil), British Council and Ministry of Education of Spain under Acciones Integradas grant no. 1016, and also UK EPSRC under grant no. K19723

[1] Department of Computer Science, mmf,npt@dcs.ed.ac.uk
[2] Department d'Arquitectura de Computadors, josepll@ac.upc.es

and data is transferred between registers and memory using load and store instructions. The lifetime of a value is the time span from the reservation of a register to hold the value up to the last moment when the value is used. Lifetimes often exceed the initiation interval, which means multiple live values from a single instruction must coexist. Early designs proposed alternative register file organizations to deal with the problem. The Polycyclic architecture ([9]) uses a delay element, implemented as a queue with shift capabilities between every pair of communicating functional units, often resulting in a full cross-bar. This queue organization facilitates write operations by means of a write pointer and compacting non-empty locations, however it requires a book-keeping function to determine the exact address of a value being read. The Cydra 5 architecture ([12]) relies on a large number of registers and provides a mechanism to perform a sort of register renaming, which also helps to avoid code size explosion, a scheme called *rotating register file*. It requires a bank of registers between every pair of communicating functional units, which also leads to a full cross-bar. In addition to the problem of overlapped lifetimes, advances in technology have increased the parallelism available in a microprocessor through a larger number of functional units, which in turn increases register pressure dramatically [7], requiring once again new register file organizations. Assuming that a single register file is not able to support the high register pressure generated by modulo scheduled loops for large numbers of functional units, we believe that some sort of register file partitioning might be a reasonable alternative. Thus, a processor composed of clusters of functional units and private register files could be used as a starting point for a new hardware scheme. However, simply reorganizing the processor in this way can not guarantee a solution for the whole problem as inter-cluster communication delays can impose a severe performance penalty. To effectively take advantage of this concept a more elaborated register file organization and scheduling mechanism should be employed.

In a modulo scheduled loop the register values used to hold data referring to the same operation in different loop iterations have the same lifetime, but with the start times offset by the initiation interval. Therefore, if two computations produce values with lifetimes of equal length, and their start times are different, then the production order of their respective values will exactly match the consumption order of the values. Under this condition the computations can name a shared queue as the common destination for their result values. Thus, sets of lifetimes of the same length could be stored in the same queue, simplifying register access and reducing register name pressure. Further investigations have shown that this constraint can be relaxed under certain strictly defined conditions to permit lifetimes of *different* lengths to share the same output queue.

We are currently investigating the possibility of designing a scalable VLIW architecture comprising clusters of functional units and private register files implemented as queue structures, which in turn may also be used for inter-cluster communication. As the number of queues will be finite the code partitioning and scheduling process will involve an element of queue allocation similar in some ways to conventional register allocation. Overall, we believe that the use of queues has

distinct advantages in terms of hardware complexity, silicon area, name space, and scalability. This paper presents the current status of our research, together with some of our initial experimental results and conclusions.

2 Using Queues to Organize Register Files

We show in [4] that the register file area needed to store enough registers and to provide sufficient access to those registers in a software pipelined loop is proportional to the cube of the number of functional units. This result clearly shows that is impractical to rely on a large multi-port register file to hold live values in a VLIW machine using modulo scheduling techniques if scalability of parallelism is the goal. It may even be the case that a shared multi-ported register file is not the most area-efficient storage scheme for the moderate degrees of ILP found in superscalar microprocessors.

This paper proposes a partitioned register file in which individually addressable registers are replaced by queues. In terms of similarities with other systems we understand that it resembles the Polycyclic machine only in which concerns writing values to a queue. The rotating register file employed by the Cydra 5 architecture could be viewed as a queue organization in which every distinct lifetime is allocated to a distinct queue, however that would require an unacceptable number of machine resources. The remainder of this paper is devoted to demonstrating that queues can reduce the register pressure generated by modulo scheduled loops in a VLIW machine, incorporating the following advantages over conventional organizations:

- *Hardware complexity and silicon area:* The access to a queue of registers is simpler than the access to a conventional register file as there is no need to select the register to be read or written to. Instead a value is always written on the last position in the queue and read from the first position, which can be controlled by means of two pointers. We expect that this organization might reduce the hardware complexity, and consequently the silicon area required.
- *Name Space:* We show in [4] that the number of registers required by a modulo scheduled loop is proportional to the number of functional units and to the pipelining degree, which increases the pressure on the name space as the machine scales up. In our queue register file model a data value is not allocated to a specific register location but instead to a specific queue, which implies that the name space problem is shifted from distinct register locations to distinct queues. We have found through experimental analysis that using a queue register file may reduce dramatically the pressure on the name space, as shown in Sect. 4.
- *Register Allocation:* The problem of register allocation, either considering a conventional register file [10] or a partitioned one [6] has been pointed out by several authors as being a non trivial task. We have developed a simple and efficient strategy to allocate data values to queues that we understand as being simpler than most of the techniques described in the literature.

- *Code Generation:* Kernel-Only code is a scheme that avoids code size explosion [11], which may be implemented if a queue register file is used along with support for predicate execution.
- *Inter-Cluster Communication:* It is well known that the efficiency of the inter-cluster communication system is a major issue to be addressed when designing clustered architectures. We believe that register queues may be used for this purpose, implementing a sort of asynchronous communication between clusters, with no need of extra instructions to move data values.

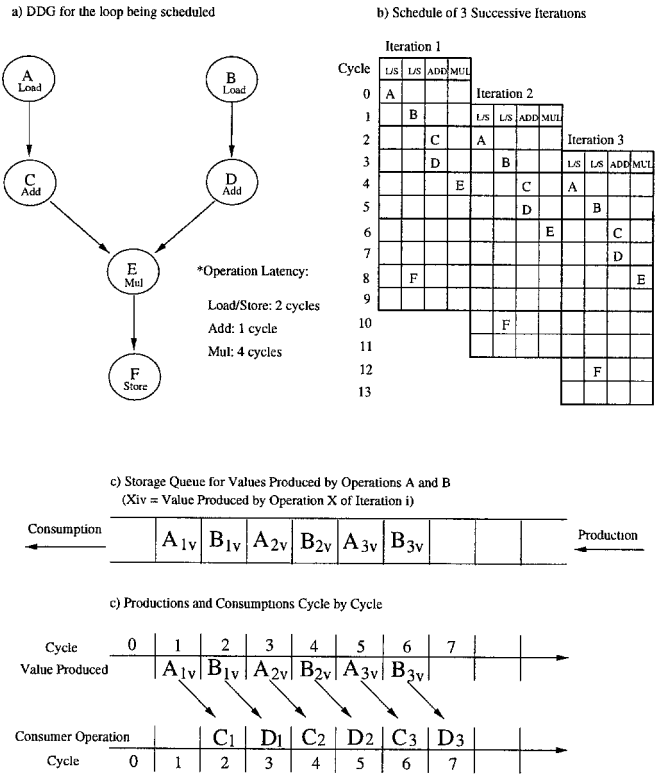

Figure 1. Allocating Registers to a Queue

To illustrate some of the ideas presented in this section we take the *data dependence graph (ddg)* of a given innermost loop (Fig. 1a) and the corresponding modulo schedule for 3 successive loop iterations (Fig. 1b). Assuming that a queue register file is being used, Fig. 1c shows the data flow in one of the storage queues, which contain values produced by successive executions of operations A and B. It can be seen that the production order of such values matches the consumption order required by operations C and D, i.e., the first element in the queue is always the value required by the next operation to be executed.

3 Queue Compatibility Condition

The ability to minimize the number of queues required by a modulo scheduled loop is critical to the use of a queue register file. We have developed a condition to check if two lifetime values can share the same storage queue. We also show how this condition can be evaluated through a simple and practical compile-time test. Due to space limitations we have ommited the theorem proof, which can be found in a technical report ([4]).

In a modulo-scheduled loop each computation generates a new value every Initiation Interval (II) cycles. Each value has a fixed lifetime which begins at some start-point and terminates at some end-point within the schedule.

Definition 3.1 (Lifetimes). On each iteration of a loop every computation **a** produces a new value which exists over a period defined by the pair $\langle S_a, S_a + L_a - 1\rangle$, where S_a is the start-point and $S_a + L_a - 1$ is the end-point of that value. We say that L_a is the lifetime of computation **a**.

Definition 3.2 (Q-compatibility). Let two computations **a** and **b** have start-points S_a and S_b, and have lifetimes L_a and L_b such that $L_a \geq L_b$. The values produced by **a** and **b** can share the same destination queue if the relative order in which they produce values is identical to the relative order in which those values are consumed by their successor computations, and their start-points are different.

It is now necessary to formulate a simple way of determining the compatibility of any pair of computations. We do this by formulating a proposition which encapsulates our definition of Q-compatibility and then we prove that there exists a simple relationship between lifetimes, start-points and Initiation Interval which can be used in a scheduler to determine Q-compatibility. We now formulate a proposition based on Definition 3.2 which provides us with a formal criteria for queue compatibility.

Proposition 3.3. *The two computations* **a** *and* **b** *are Q-compatible if, and only if:*

$$\forall_{i,j\geq 0} : a_i > b_j \Rightarrow a_i + L_a > b_j + L_b \quad (1)$$

$$\wedge\ a_i < b_j \Rightarrow a_i + L_a < b_j + L_b \quad (2)$$

$$\wedge\ a_i \neq b_j \quad (3)$$

This proposition, although an accurate formulation of Definition 3.2, cannot be used directly when scheduling a loop as it contains universal quantifiers. These imply a large, possibly unbounded, search space for i and j. The following theorem defines an alternative, and computationally efficient, test for Q-compatibility.

Theorem 3.4 (Exact Compatibility Test:). *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.*

4 Experimental Evaluation

In order to obtain quantitative data regarding modulo scheduled loops for a hypothetical VLIW machine, an experimental scheduling framework has been built. The basic algorithm used in this framework is *Iterative Modulo Scheduling (IMS)* [8]. The scheduler assumes the existence of a simple VLIW machine, comprising of some fully pipelined functional units connected to either a *multi-ported register file (RF)* or a *register file organized by means of queues (QRF)*. Three machines configurations have been considered, as shown in Table 1.

To evaluate the effectiveness of queues as an alternative to conventional registers all eligible innermost loops from the Perfect Club Benchmark were scheduled, totalling 1258 loops. The optimizations and data dependence analysis were performed by the ICTINEO compiler [1], which supplied the input data set used by our framework. Due to space limitations we only briefly present some of the experimental results obtained, which can be found in [4].

Functional unit type	Operation latency	Issue rate	Number of functional units		
			machine A	machine B	machine C
load/store	2	1/~	2	2	4
add/subtract	1	1/~	1	2	4
multiply	4	1/~	1	2	4

Table 1. Functional units for three target machine configurations

Number of Queues Required The graphics presented in Fig. 2 shows the fraction of loops, from the set of 1258 loops considered, that can be scheduled employing only a given number of queues. The results show that with a fixed number of 32 queues it is possible to schedule most of the loops regardless the number of functional units, suggesting that number as being the size of the name space required, which is considerably smaller than that required by conventional register file organizations. It also shows a tolerable increase in the required number of queues as more functional units are used, suggesting that the scalability of the model is not constrained by this resource.

Number of Storage Positions Required In Fig. 3 it is shown the total number of queue positions required to schedule a given fraction of the loops. It can be seen that it is possible to schedule over 90% of all the loops using no more than 64 queue positions. It may be worth at this point to make a rough comparison between this figures and the register requirements when using a conventional register file organization. Similar analyses performed by other groups [7, 3, 5] found that it is possible to schedule around 90% of all the loops with 32 registers, which may suggest that their schemes are more efficient regarding this aspect. In spite of our beliefs that the possibly lower complexity of a queue register file may compensate this difference, we are currently working in a number of alternatives to improve this figure.

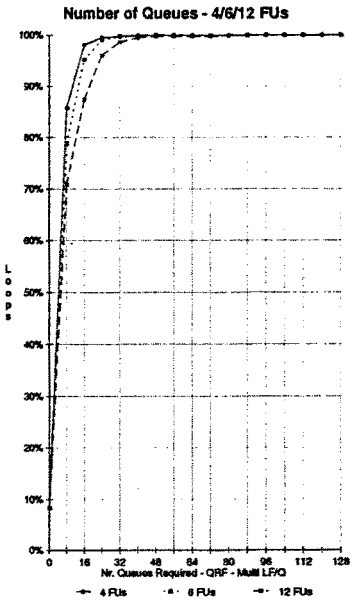

Figure 2. Number of Queues Required

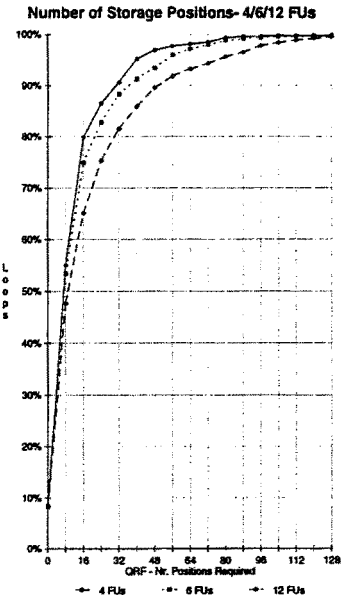

Figure 3. Queue Capacity Required

Loops that Benefit from Greater Parallelism We found that significant speedups can be attained for around 70% of the loops when more functional units are employed, which justify the use of aggressive hardware configurations. In most of the cases the number of extra queues required for that falls between 0 and 15, which we understand as being a good prospect in terms of scalability.

5 Conclusions

We have investigated alternative register file organizations to address the high register pressure generated by a modulo scheduled loop. A register file organized by means of queues has been considered, and a number of quantitative data regarding machine resources was obtained from a preliminary evaluation. We have observed that the number of distinct queues required to schedule the benchmark loops is around 32 for configurations up to 12 functional units, which is less than other schemes reported in the literature. We have found that the total number of bits of queue storage is larger than that required by a conventional register file but we believe that the silicon area requirements will remain significantly lower. The small differences found between machine resources required by distinct number of functional units suggests that there is an advantage in terms of scalability, which is not the case of systems that relies on conventional register files or cross-bar organizations.

We are currently working in a number of improvements on the proposed model, including loop unrolling to maximize functional units utilization, introduction of copy operations to deal with the problem of simultaneous writes of the same value to distinct queues, allocation of loop invariant and a hardware complexity model for the queue register file. We are also working on a new machine model organized by means of clusters composed of functional units and a private register file, which in turn communicate among each other through a bidirectional ring of queues. Finally, an enhanced machine model should be employed in the near future, increasing the level of details and assuming a finite number of machine resources, which may lead to the use of other techniques like graph coloring and the introduction of spill code.

References

1. E. Ayguadé, C. Barrado, J. Labarta, J. Llosa, D. Lopez, S. Moreno, D. Padua, E. Riera, and M. Valero. Ictineo: Una herramienta para la investigacion en paralelismo a nivel de instrucciones. In *VI Jornadas de Paralelismo*, July 1995.
2. A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9), 1981.
3. A. Eichenberger, E. Davidson, and S. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of the MICRO-27 - The 27th Annual International Symposium on Microarchitecture*, November 1994.
4. Marcio M. Fernandes, Josep Llosa, and Nigel Topham. Using queues for register file organization in VLIW architectures. Technical Report ECS-CSG-29-97, Edinburgh University, February 1997.
5. R. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the SIGPLAN'93 - Conference on Programming Language Design and Implementation*, 1993.
6. J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, November 1995.
7. 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*, October 1994.
8. B. Rau. Iterative modulo scheduling. *The International Journal of Parallel Processing*, February 1996.
9. 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*, October 1981.
10. B. Rau, M. Lee, P. Tirumalai, and M. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 - Conference on Programming Language Design and Implementation*, June 1992.
11. B. Rau and P. Tirumalai M. Schlansker. Code generation schema for modulo scheduled loops. In *Proceedings of the MICRO-25 - The 25th Annual International Symposium on Microarchitecture*, December 1992.
12. B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer. *Computer*, January 1989.