Computer Systems Group

Using Queues for Register File Organization in VLIW Architectures

by

Marcio Merino Fernandes and Josep Llosa and Nigel Topham

CSG Report Series **ECS–CSG–29–97**

Computer Systems Group **February 1997**

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Using Queues for Register File Organization in VLIW Architectures

Marcio Merino Fernandes* and Josep Llosa† and Nigel Topham‡

Technical Report ECS–CSG–29–97
Department of Computer Science
University of Edinburgh

February 17, 1997

Abstract: Software pipelining is an effective technique for increasing the throughput of loops in superscalar or VLIW machines. However, software pipelining generates high register pressure, which in some cases requires the introduction of spill code into the schedule. This report shows that large multi-ported register files present significant problems in the construction of scalable VLIW systems. In an attempt to address this problem we are investigating the possibilities for VLIW 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 report we present an experimental evaluation of the machine resources required to support modulo scheduling under a variety of VLIW register file configurations. The results obtained suggests that the use of queues is a feasible alternative to global register files.

Keywords: Software pipelining, instruction-level parallelism

*e-mail: mmf@dcs.ed.ac.uk
†e-mail: josepll@ac.upc.es
‡e-mail: npt@dcs.ed.ac.uk

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. Rau and Fisher define a hierarchy of ILP architectures based on the ways in which ILP is extracted [24]. In their Sequential Architectures the program contains no explicit information regarding dependencies that exist between instructions, so dependencies must be determined by hardware at run-time using only sequentially ordered operations that are in flight (those that have been issued but have not completed yet). Superscalar processors are representative of this class of architecture [12]. In Independence Architectures 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. If the compiler can also provide information about functional unit and issue cycle (i.e. scheduling of operations), then the hardware does not take any decision at run time. Very Long Instruction Word (VLIW) processors are representative of this type of architecture. The ideal VLIW machine has a number of concurrent functional units (FU), fed by a register set that can issue two operands (reads) and perform one write operation per functional unit in each cycle [5]. The development of such machines is justified by a number of studies into the available ILP in selected programs. These generally conclude that there is sufficient parallelism, particularly in scientific applications, to warrant the exploitation of ILP [25].

The scheduling of operations plays a major role in achieving near optimal performance from an ILP machine. Often this procedure relies on specific transformations performed at early compilation stages [4] and is supported by special purpose hardware features [23]. Scheduling algorithms for ILP can be classified according to the nature of the control flow graph being scheduled [24]. The control flow graph may consist of a single basic block or multiple basic blocks, which in turn can be cyclic or acyclic. Local scheduling algorithms work only on acyclic basic blocks, and generally produce very efficient schedules. However the small size of typical basic blocks, together with data dependencies and execution latencies constitute an unacceptable barrier to ILP using purely local scheduling.

Acyclic global scheduling algorithms consider more than one basic block at a time, working either with acyclic control flow graphs (graphs that contain no cycles or cyclic graphs for which a self-imposed barrier exists at each back edge) or with cyclic ones (graphs containing loops but not considering a back edge barrier). The class of cyclic global scheduling algorithm tries to find ILP in loops by overlapping the execution of multiple basic blocks representing multiple iterations of the same loop body Some approaches use loop unrolling, a scheme that replicates the body of a loop a number of times and then perform some local scheduling over the unrolled loop body. An alternative approach, known as software pipelining, was proposed by Charlesworth [6], with the objective of initiating successive loop iterations before prior iterations had completed. Several software pipelining schemes that have been proposed over the past few years [1]. Modulo scheduling is a class of software pipelining algorithms in which all loop iterations have the same schedule of operation [21]. The number of cycles between the initiation of successive iterations is called the initiation interval (II). Naturally, a common schedule must be found for all iterations such that no resource usage conflicts arise and all instruction dependencies are satisfied. The minimum initiation interval (MII) is a lower bound on II that can normally be determined analytically according to machine resources and data dependence constraints. The actual II may be higher than MII in practice, and as the problem of finding an optimal software pipelining schedule is known to be NP-complete [14] any practical algorithm must rely on heuristics.

Most software pipelining schemes assume an architectural model in which arithmetic operations are all register-register operations 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. This increases register requirements dramatically [18, 19]. The register file of a typical microprocessor around 1996-97 has 32-64 general purpose float-point registers with 6-12 ports. This configuration is enough in many cases, but is far from ideal in some others, and sometimes the only alternative is to add spill code to the schedule, with an obvious penalty to be paid in performance [17]. Some highly multi-ported register files have been built [13], but this does not appear to be a good long term solution for the problem, as the area of a conventional register file is proportional to the square of the number of ports and to the cube of the number of functional units. Some algorithms developed recently take this into account and include heuristics in order to minimize the number of registers required [10, 16]. Although successful when compared against previous algorithms, it seems that another approach is necessary, especially if scalable VLIW is an objective.

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. The renaming which results from rotating register files is implicit in a queue based scheme. 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 hardware complexity, silicon area, name space, and scalability.

This technical report presents the current status of our research, together with some of our initial experimental results and conclusions. In Section 2 we present the most common terminology associated with modulo scheduling and a simple example to illustrate concepts involved and show how register requirements can be estimated. Then we present some of the alternatives found in the literature to address the problem of register pressure. These alternatives include scheduling algorithms employing heuristics to minimize register requirements, as well as more complex register file organizations. Finally we present our approach to the problem. In section 3 we present a formal condition under which two lifetime values can share a common storage queue, derive an efficient compile-time test, and prove the correctness that the test. In order to evaluate our approach to the problem we have carried out a number of experiments, the main results of which are presented in Section 4. Finally Section 5 contains conclusions and future directions for this work.

2 Register Pressure in Modulo-scheduled Loops

In general the parallelism required by a high performance ILP machine can only be achieved if instructions belonging to distinct basic blocks are executed simultaneously. In a modulo scheduled loop that means simultaneous execution of instructions from distinct loop iterations. Prior to scheduling, a loop is typically represented by a data dependence graph (DDG) whose nodes represent atomic operations (e.g. load, store, add, mul). A directed edge between two nodes defines a dependence, meaning that the operation at the target node depends on the execution of an operation at the source node. These dependencies can be either control or data dependences [26]. Control dependences can be eliminated through IF-conversion techniques, in which case they become data dependences [2]. A data dependence typically implies the need for a physical register to store a data value produced by the source node and consumed by the target node. Each dependence edge has the attribute of distance, representing the number of iterations separating the dependent operations. A distance of zero means that both operations belongs to the same iteration and this we call an intra-iteration dependence. If dependence distance has positive value i the dependent operation (target node) depends on the source node operation from the ith previous iteration, and is known as an inter-iteration flow dependence. Some authors include a second edge attribute representing delay, which states the minimum time interval that must exist between the start of both operations. We have opted to not include this attribute in the dependence graph as it depends on the function unit latencies of the specific machine configuration in use. Instead we calculate it for each particular case.

To illustrate those concepts Example 1 shows a modulo scheduled loop for a machine model comprising 4 FUs, as defined in Table 1. The data dependence graph of Figure 1 shows the machine instructions of the simple innermost loop used in this example.

Function unit	Number	latency (cycles)	issue interval (cycles)
load/store	2	2	1
add/subtract	1	1	1
multiply	1	4	1

Table 1: Example machine model

In Figure 2 we show the schedule of operations for a single iteration i of the loop, which takes 11 cycles to complete. The lifetime values associated with each operation are represented by bars in the diagram. We assume that the lifetime of a value starts when the producer is issued and ends in the cycle prior to the issue of the instruction which represents the last use of the value (last consumer).

Figure 3a shows the schedule for 5 successive loop iterations. It should be noticed that a new loop iteration starts every two cycles. When the 5th iteration starts the schedule reaches its steady state, with one iteration starting and one iteration finishing every II cycles. The performance gain obtained by this technique can be seen when the execution

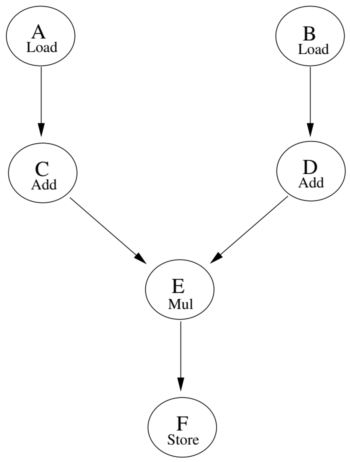


Figure 1: Dependence graph for Example 1

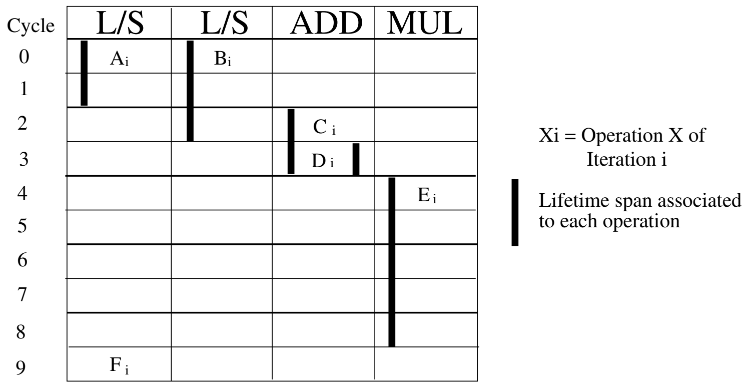


Figure 2: Schedule of operations for Example 1

reaches the steady state: the execution of a single loop iteration in a machine unable to exploit ILP would take 11 cycles to complete, but the machine model of this example requires only 2 cycles. In steady state the same schedule of operations is repeated every II cycles, and a compact notation for it, the kernel code, is presented in Figure 3b. The subscript of each operation indicates the iteration to which it belongs. The code prior the steady state is named prologue and the code which follows the kernel is known as the epilogue. Because a lifetime value may span more than II cycles, successive values produced by one operation overwrite previous ones before their last use, thus they must be allocated to distinct storage locations. For instance, the register used by operation E remains alive for 5 cycles, but a new instance of operation E starts every 2 cycles, requiring distinct storage locations for the values produced by every 3 consecutive instances of operation E. A compact notation to describe the maximum number of live registers at any time MaxLive is shown in Figure 3c. In this particular example it can be seen that MaxLive = 7, indicating that this schedule requires at least 7 physical registers.

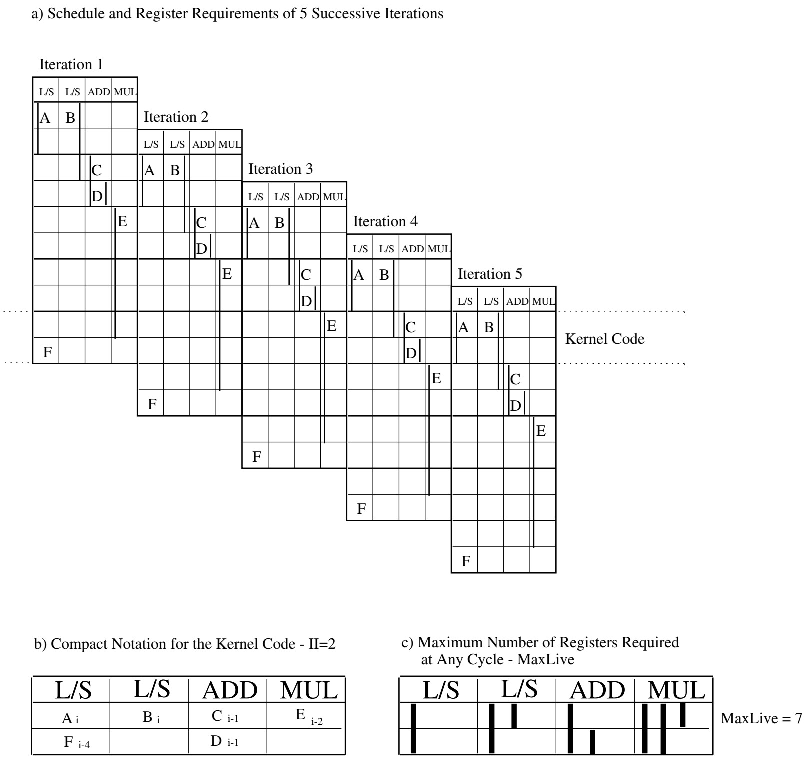


Figure 3: Register requirements for Example 1

2.1 Lifetime Sensitive Modulo Scheduling Algorithms

The goals of the first modulo scheduling algorithms were to produce optimal schedules (i.e. II=MII in most of the cases) and be computationally tractable, usually relying on heuristics to attain those objectives [20]. Several studies have pointed out that those scheduling techniques produce high register preassure [18, 19, 8, 9], and recent scheduling algorithms take that into account by using heuristics that are sensitive to lifetimes. One approach to the problem is the Stage Scheduling heuristics [7], whose basic idea is to move scheduled operations by multiples of II cycles. The Slack Scheduling [10] proposed by Huff schedules some operations early and others late, in an attempt to reduce MaxLive. Another algorithm, Swing Modulo Scheduling [16], performs a node ordering phase before scheduling, taking into account possible recurrences and critical paths associated with each node. Experimental evaluations have shown that those techniques are effective at reducing register pressure. An example of those technique is shown in Figure 4, which is a modification on the schedule presented in Example 1. In this case operation B was delayed by 1 cycle, and operation F was antecipated by 1 cycle, which reduced the value of MaxLive from 7 to 6, as seen in Figure 5, without causing any performance degradation; the value of II is unchanged. This example shows that it is possible to have more than one valid schedule for the same loop, all of them with identical execution rates, but requiring different numbers of registers.

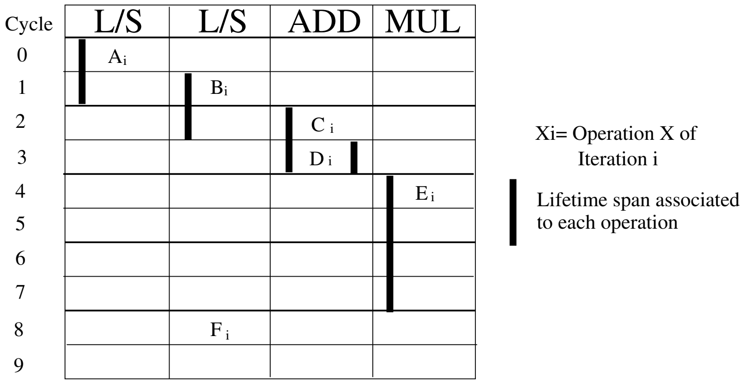


Figure 4: Alternative Schedule of operations for Example 1

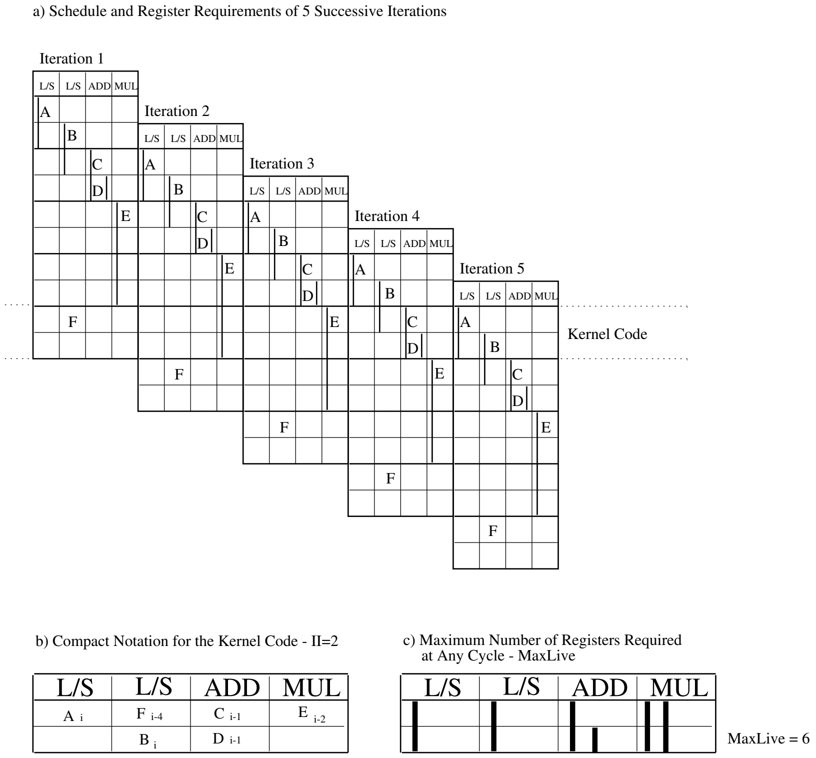


Figure 5: Alternative register requirements for Example 1

2.2 Multi-ported Register Files

Although lifetime sensitive algorithms are effective at reducing register pressure, the number of register required by some loops is still much higher than the number of available registers. Experimental results obtained by Llosa [18] show that only a small fraction of the loops he considered (about 5%) required a large number of registers, however they represent a significant slice of the total execution time (30%), which indicates that they are critical to the overall system performance. These are loops with large numbers of operations, a high degree of intrinsic ILP, and hence large numbers of live values. When scheduled on a processor with a large number of functional units their throughput is high, but at the cost of a large register requirement. A typical fifth-generation RISC processor has at least 32 architected registers contained within a common register file shared by 3-5 functional units through a relatively small number of access ports (typically 6-12). As the number of functional units in future generations, the register requirements will grow, both in the capacity and also in the number of ports. We believe that the design of a scalable VLIW machine cannot rely on a large multi-ported register file; the following section explains why.

2.2.1 Register File Area

Let us assume that a p ported register file, containing r registers is fully connected to a collection of f functional units, each having a latency of l cycles.

The silicon area requirement for such a register file is shown diagramatically in Figure 6. For some constant value K, the area A of the register file is given by:

$$A = Krdp^2 = \Theta(rp^2) \tag{1}$$

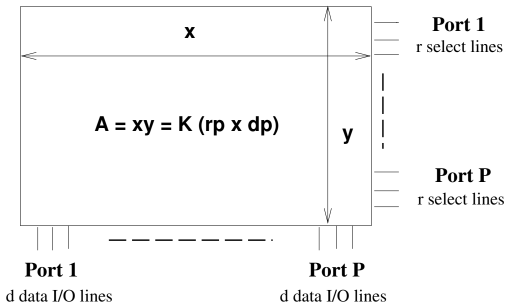


Figure 6: Multi-ported register file layout

In a modulo scheduled loop each operation in flight reserves n register names, where n is the number of software pipe-stages straddled by each lifetime. The number of operations in flight is determined by the product of the instruction issue width and the pipeline lengths. This equals the number of independent functional units, f, multiplied by the number of pipeline stages of each FU, l. Thus, the number of registers r required to execute a modulo scheduled loop in a VLIW machine is:

$$r = nfl = \Theta(fl) \tag{2}$$

To sustain an average issue rate of s instructions per cycle it is necessary to have a register bandwidth of at least $2s$ reads and s writes per cycle, requiring 2 read ports and 1 write port per instruction. Under the reasonable assumption that $s = \Theta(f)$ we can say:

$$p = \Theta(f) \tag{3}$$

From equations 2 and 3 the register file area needed to store enough registers and to provide sufficient access to those registers in a software pipelined loop can be expressed in terms of the number of functional units:

$$A = \Theta(lf^3) \tag{4}$$

Thus we have found that the area of such register file 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.

2.3 Partitioned Register Files

Partitioned register files have been proposed as an alternative to a large multiport register file. Capitanio proposes a model in which clusters of FUs are connected to private small register files [5]. One drawback in this approach is that inter-cluster communication is accomplished through the main memory system, which requires additional Load/Store operations. The Transport Triggered Architecture also employs partitioned register files [11] and uses heuristics to minimize access conflicts. It has been ackowledged by several authors that a partitioned register file has a number of advantages over a large multiport register file, such as savings in silicon area, reduction in access time, low complexity, and scalability. However a partitioned register file imposes further constraints on both the scheduler and the register allocator, which may result in performance degradation if not properly handled.

2.4 Using Queues to Organize Register Files

This report proposes a partitioned register file in which individually addressable registers are replaced by queues. The remainder of this report is devoted to demonstrating that queues can reduce the register pressure generated by a modulo scheduled loops in a VLIW machine. The partitioning process itself incorporates the advantages pointed out in Section 2.3, and the use of queues results in the following extra benefits:

- *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. Instead a value is always written on the last position in the queue and read from the first position, eliminating the need of additional logic controls to select intermediate registers. That might reduce the hardware complexity, and consequently the silicon area required by such organization. The inclusion of extra registers in the queue should not cause any increase in the control logic, which is desirable if the model scales up.

- *Name Space:* As shown in Section 2.2.1 the number of registers required by a modulo scheduled loop is proportional to the number of functional units and to the pipelining degree. Thus it is expected high pressure on the name space to occurs 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 shifited 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 Section 4.

- *Register Allocation:* The problem of register allocation, either considering a conventional register file [22] or a partitioned one [11] 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. The scheme relies on a queue compatibility condition under which two lifetimes can share the same storage queue. This condition and the corresponding proof is presented in Section 3. We have also performed a number of experiments using this register allocation strategy, as seen in Section 4.

- *Inter-Cluster Communication:* As already said we consider the possibility of organizing a VLIW machine in clusters of functional units and private register files. It is well known that the efficiency of the inter-cluster communication system is a major issue to be addressed in any clustered architecture. We believe that register queues may be used for this purpose, implementing a sort of assynchronous communication between clusters, with no need of extra instructions to move data values. The efficiency of such machine organization is highly dependent on the code partitioning process, which we should investigate in the near future.

We think that those advantages are enough to justify the investigation of such machine organization, which is currently been carried on in addition to the work developed so far.

To illustrate the ideas presented in this section we take 3 successive loop iterations of the alternative modulo schedule produced for Example 1 (Figure 7a) and show a possible register allocation to a hypothetical queue register file. In Figure 7b we show one of the storage queues used, which contain values produced by successive executions of operations A and B. It can be seen in Figure 7c 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. It should be noted that there are no more than one production and one consumption at any given cycle, assuring that no access conflicts will arise.

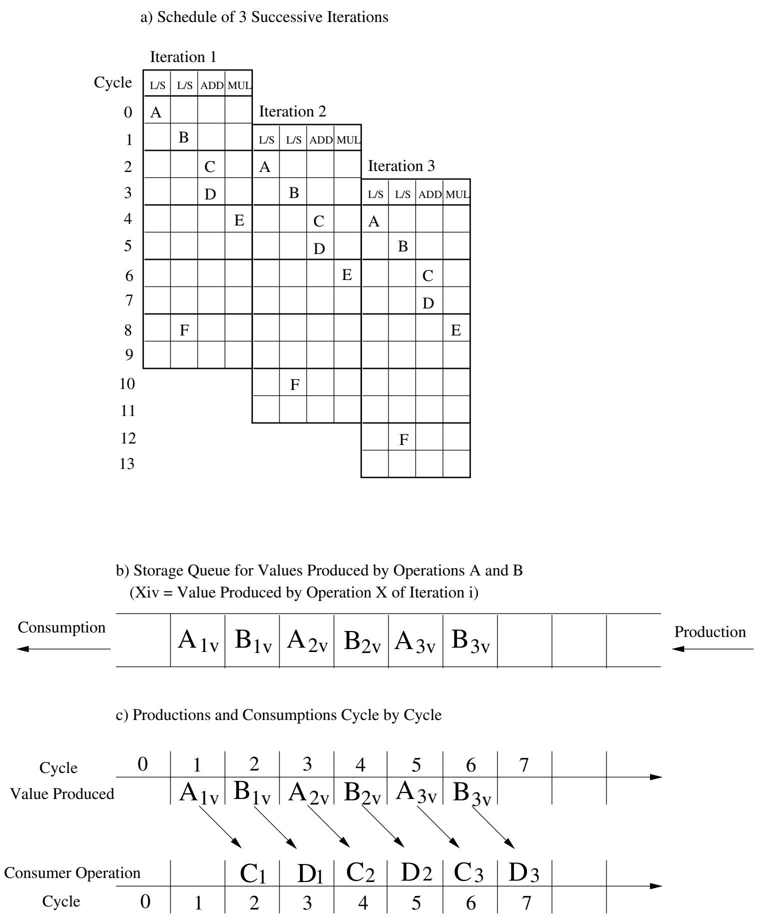


Figure 7: Allocating Registers to a Queue

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.

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 (Vector lifetimes) *In a modulo-scheduled loop every computation* **a** *produces a vector of lifetimes* A*:*

$$A \equiv \{\langle a_n, a_n + L_a - 1\rangle : a_n = S_a + n.II\}_{n\geq 0}$$

Definition 3.3 (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. The proofs we develop use modulo arithmetic and rely on the following four lemmas.

Lemma 3.1 *For all integers* x*,* y *and* n*,* $x \equiv y \Rightarrow [x]_n \equiv [y]_n$*.*

Lemma 3.2 *For all integers* x*,* y *and* n*,* $[x + ny]_n \equiv [x]_n$*.*

Lemma 3.3 *For all integers* x *and* n*,* $x > 0 \Rightarrow [x]_n \leq x$*.*

Lemma 3.4 *For all integers* x *and* n*,* $0 \leq x < n \Rightarrow [x]_n \equiv x$*.*

We now formulate a proposition based on Definition 3.3 which provides us with a formal criteria for queue compatibility.

Proposition 3.1 *The two computations* **a** *and* **b** *are Q-compatible if, and only if:*

$$\begin{aligned}\forall_{i,j\geq 0} \quad &: \quad a_i > b_j \Rightarrow a_i + L_a > b_j + L_b && (5)\\ &\wedge \quad a_i < b_j \Rightarrow a_i + L_a < b_j + L_b && (6)\\ &\wedge \quad a_i \neq b_j && (7)\end{aligned}$$

This proposition, although an accurate formulation of Definition 3.3, 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 3.1 (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* $L_a - L_b < [S_b - S_a]_{II}$.

Proof

To prove Theorem 3.1 we must demonstrate that it logically implies proposition 3.1. We can express this requirement as the propositional formula $\models P \Rightarrow Q$, where

$$\begin{aligned}Q &\equiv \forall_{i,j\geq 0} : \quad R_1 \wedge R_2 \wedge R_3 && (8)\\ R_1 &\equiv a_i > b_j \Rightarrow a_i + L_a > b_j + L_b && (9)\\ R_2 &\equiv a_i < b_j \Rightarrow a_i + L_a < b_j + L_b && (10)\\ R_3 &\equiv a_i \neq b_j && (11)\\ P &\equiv L_a - L_b < [S_b - S_a]_{II} && (12)\end{aligned}$$

We now show that this formula holds using proof by contradiction:

1. Let there exist interpretations of P and Q which render $P \Rightarrow Q$ false. Hence, there must exist values of i, j, S_a, S_b, L_a, L_b, and II such that P is true and Q is false.

2. From the definition of Q in equation (8) any one of R_1, R_2 and R_3 can be false for Q to be false. We consider these cases in steps 3, 4 and 5.

3. Let R_1 be false, then there must exist $i, j \geq 0$ such that $a_i > b_j$ and $a_i + L_a \leq b_j + L_b$. Thus:

   $$\exists_{i,j\geq 0} : \quad a_i + L_a \leq b_j + L_b$$

   and so

   $$\exists_{i,j\geq 0} : \quad L_a - L_b \leq b_j - a_i$$

   But $a_i > b_j$, so $b_j - a_i < 0$, and hence

   $$L_a - L_b \leq b_j - a_i < 0 \qquad (13)$$

   According to statement 1, P is true and therefore $L_a - L_b < [S_b - S_a]_{II}$ is also true. But the defined range of the integer modulus function is $n > [x]_n \geq 0$. We therefore have a contradiction between equation (13) and statement 1. Consequently R_1 cannot be false if P is true.

4. Let R_2 be false, then there must exist $i, j \geq 0$ such that $a_i < b_j$ and $a_i + L_a \geq b_j + L_b$. Thus:

$$\exists_{i,j\geq 0} \;:\; \quad a_i + L_a \geq b_j + L_b$$

and so

$$\exists_{i,j\geq 0} \;:\; \quad L_a - L_b \geq b_j - a_i \tag{14}$$

We know from Definition 3.2 that $a_i = S_a + i.II$ and $b_j = S_b + j.II$, so we may write:

$$b_j - a_i = S_b - S_a + II(j - i) \tag{15}$$

By Lemma 3.1 and equation (15) we may write:

$$[b_j - a_i]_{II} = [S_b - S_a + II(j - i)]_{II} \tag{16}$$

By Lemma 3.2, equation (16) can be reduced to:

$$[b_j - a_i]_{II} = [S_b - S_a]_{II} \tag{17}$$

Recall that for R_2 to be false we must satisfy the following inequalities:

$$\exists_{i,j\geq 0} \;:\; \quad L_a - L_b \geq b_j - a_i > 0$$

But since proposition P is assumed to be true, then:

$$L_a - L_b < [S_b - S_a]_{II}$$

Since both equation (14) and proposition P must both hold, we can write:

$$[S_b - S_a]_{II} > L_a - L_b \geq b_j - a_i > 0$$

Eliminating $L_a - L_b$, we get:

$$[S_b - S_a]_{II} > b_j - a_i > 0 \tag{18}$$

From equation (16) we know that $[S_b - S_a]_{II} = [b_j - a_i]_{II}$, and hence substituting for $[S_b - S_a]_{II}$ in equation (18) we get:

$$[b_j - a_i]_{II} > b_j - a_i > 0 \tag{19}$$

This contradicts Lemma 3.3 so R_2 cannot be false.

5. Let R_3 be false, then $\exists_{i,j\geq 0} \;:\; a_i = b_j$ and consequently $[b_j - a_i] = 0$. From equation (17) we can further deduce that $[S_b - S_a]_{II} = 0$. However, if as P it true we can say that:

$$L_a - L_b < [S_b - S_a]_{II} = 0$$

But as an assumption of the theorem we have:

$$L_a - L_b \geq 0$$

This represents a contradiction, so R_3 cannot be true when P is true.

We have thus demonstrated that none of R_1, R_2 or R_3 can be false if P is true, which in turn means that Q cannot be false if P is true. □

We have therefore shown that whenever the inequality in Theorem 3.1 is satisfied, the condition expressed in Proposition 3.1 is also satisfied. This provides us with a guarantee that the Q-compatibility test from Theorem 3.1 will always indicate incompatibility for a pair of lifetimes that are incompatible. We would also like to guarantee that whenever our Q-compatibility test indicates incompatibility, then so does Proposition 3.1. This would demonstrate equivalence between Theorem 3.1 and Proposition 3.1 and show that Theorem 3.1 is an exact test.

Theorem 3.2 (Exactness) *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 $L_a - L_b < [S_b - S_a]_{II}$.

Proof

To prove this theorem we must show that $\models P \Rightarrow Q$ and $\models \neg P \Rightarrow \neg Q$, for P and Q defined by equations (12) and (8) respectively. Theorem 3.1 established $\models P \Rightarrow Q$, so it only remains to show $\models \neg P \Rightarrow \neg Q$.

1. Let us assume that there exist interpretations of P and Q such that $\neg P \Rightarrow \neg Q$ is false. Therefore $\neg P$ must be true and $\neg Q$ must be false.

   Recall that Q is defined as:

   $$\forall_{i,j \geq 0} \; : \quad R_1 \wedge R_2 \wedge R_3$$

   Hence $\neg Q$ can be written:

   $$\exists_{i,j \geq 0} \; : \quad \neg R_1 \vee \neg R_2 \vee \neg R_3$$

   For $\neg Q$ to be false we must not be able to find values of i and j which satisfy $\neg R_1 \vee \neg R_2 \vee \neg R_3$. In statements 2, 3 and 4 we consider three cases which cover all possible relative values of a_i and b_j. These are $a_i < b_j$, $a_i > b_j$, and $a_i = b_j$.

2. Assume $a_i < b_j$, then R_1 and R_3 are both true. However, R_2 is only true if:

   $$\forall_{i,j \geq 0} \; : \quad a_i + L_a < b_j + L_b \tag{20}$$

   Under these conditions we can assert:

   $$\forall_{i,j \geq 0} \; : \quad L_a - L_b < b_j - a_i \tag{21}$$

   and

   $$\forall_{i,j \geq 0} \; : \quad b_j - a_i > 0 \tag{22}$$

   From the definition of $\neg P$ we know that:

   $$L_a - L_b \geq [S_b - S_a]_{II} \tag{23}$$

   Substituting for $[S_b - S_a]_{II}$ according to equation (17) in equation (23) we get:

   $$L_a - L_b \geq [b_j - a_i]_{II} \tag{24}$$

Combining equations (24), (21) and (22) yields the following proposition:

$$\forall_{i,j\geq 0} \ : \quad b_j - a_i > L_a - L_b \geq [b_j - a_i]_{II} > 0 \tag{25}$$

However, every computation in a schedule is executed exactly once per II cycles when the loop is in kernel mode (i.e. after the pipeline is primed, but before the shutdown phase). Consequently it is axiomatic that:

$$\exists_{i,j\geq 0} \ : \quad b_j - a_i < II \tag{26}$$

Equations (25) and (26) tell us that there exist i and j such that:

$$II > b_j - a_i > 0$$

Hence, by Lemma 3.4 we can assert:

$$[b_j - a_i]_{II} \equiv b_j - a_i \tag{27}$$

Substituting for $[b_j - a_i]$ in equation (25) using equation (27) yields:

$$b_j - a_i > b_j - a_i$$

This is a contradiction and so there exist values of i and j for which R_2 cannot be true when $\neg P$ is true and $a_i < b_j$. Under these conditions $\neg R_2$ cannot be false and consequently $\neg Q$ cannot be false.

3. Assume $a_i > b_j$, then R_2 and R_3 are both true. However, R_1 is only true if:

$$\forall_{i,j\geq 0} \ : \quad a_i + L_a > b_j + L_b \tag{28}$$

Under these conditions we can assert:

$$\forall_{i,j\geq 0} \ : \quad L_a - L_b > b_j - a_i < 0 \tag{29}$$

From the definition of P in equation (12) we know that $\neg P$ is given by:

$$\neg P \quad \equiv \quad L_a - L_b \geq [S_b - S_a]_{II} \tag{30}$$

Substituting for $[S_b - S_a]_{II}$ in equation (30) using equation (17) we get:

$$L_a - L_b \geq [b_j - a_i]_{II} \tag{31}$$

As $a_i > b_j$, we know $[b_j - a_i]_{II} \neq 0$, so we may write:

$$L_a - L_b \neq 0 \tag{32}$$

It is an assumption of the theorem that $L_a \geq L_b$. Let us therefore assume $L_a = L_b$, and thus $L_a - L_b = 0$. This directly contradicts equation (32) and we may conclude that if $\neg P$ is true when $a_i > b_j$ then R_1 must be false. Hence $\neg R_1$ must be true and $\neg Q$ cannot be false.

4. Assume $a_i = b_j$. Then R_1 and R_2 are both true but R_3 is false. Hence $\neg Q$ is true. Thus when $a_i = b_j$ and $\neg P$ is true, $\neg Q$ is always true.

5. We have shown that whatever the relative values of a_i and b_j it is not possible for $\neg Q$ to be false when $\neg P$ is true. Thus $\models \neg P \Rightarrow \neg Q$. □

4 Experimental Evaluation

In this section we present some of the experimental results obtained during the development of this work. First the experimental framework used in the experiments is described, then we explain the purpose and simplifying assumptions of the experiments. Finally the machine models considered and the corresponding quantitative figures are shown, along with some concluding remarks.

4.1 Experimental Framework

In order to obtain quantitative data regarding modulo scheduled loops for a hypothetical VLIW machine, an experimental scheduling framework has been built, using C++ language and LEDA library routines. The main characteristics of the framework are:

- *Target machine model:* The scheduler assumes the existence of a simple VLIW machine, comprising of some fully pipelined functional units connected to a *multiported register file (RF)*, thus being able to have simultaneous access to it. Alternatively it is assumed the existence of a *register file organized by means of queues (QRF)*. It should be noticed that the framework can adapt to distinct machine configurations, varying parameters such as number of functional units, pipeline degree, register file organization, and register allocation policy.

- *Modulo Scheduling Algorithm:* Rau's *Iterative Modulo Scheduling (IMS)* [20] is used as the basic algorithm in this framework. It repeatedly schedules and reschedules operations in search for a "fixed point" solution that satisfies all the scheduling constraints. If the search fails to yield a valid schedule even after a large number of steps, it is assumed that no feasible schedule exist at this II, and so its value is increased. The number of scheduling steps performed for a given II defines and control the amount of backtracking allowed.

- *Input data:* Innermost loops, comprising of a set of instructions, latencies and data dependences among them.

- *Output data:* Corresponding modulo schedule and information about register usage, lifetimes, number of queues, and other quantitative figures, as shown in Section 4.5.

4.2 Schedule Analysis

Along with the modulo schedule corresponding to the input loop, the experimental framework can also provide information regarding the schedule effectiveness and resources usage, as shown bellow:

- *Initiation Interval and Related Parameters:* As already said, the II is the number of cycles between the initiation of two successive iterations in a software pipeline schedule, and it is the ultimate performance measure for a modulo schedule as it is directly related to its execution time. The **Minimum Initiation Interval (MII)** achievable depends on resource constraints (**ResMII**) and recurrence constraints (**RecMII**), being the maximum of these two values. ResMII is calculated according to the machine resources required by the loop, thus the larger the machine resources, the smaller the ResMII. On the other hand, RecMII is not dependent on machine resources but on recurrences found in a loop. A loop contains a **recurrence** if an operation in one iteration has a dependence upon the same operation from a previous iteration. RecMII is calculated by totalling, for each recurrence, the minimum number of cycles that two iterations must elapse in order to preserve all the dependences found in a recurrence circuit. This imposes a more severe constraint on the MII, since it cannot be changed simply by employing a more sophisticated machine. As its name implies, the MII indicates the minimum value that II can assume, but there is no guarantee that a valid schedule exists for that, although in practice that has been achieved for most of the loops when using some of the recently developed algorithms reported in the literature. Because of that, a comparison between MII and II constitutes a good indication of how well a scheduler performs its functions, which is done by the experimental framework. **Schedule length** accounts for the number of cycles necessary to execute all of the instructions of a single loop iteration. **Stage Count** is the schedule length divided by the II, and indicates the number of distinct iterations that are been executed at the same time when the schedule is in the steady state phase. The stage count should be minimized whenever it is possible in order to avoid long and unefficient prologue and epilogue phases, and also to reduce register pressure.

- *Register Usage - Conventional RF Organization:* Assuming the use of a RF, it is estimated the maximum number of registers (MaxLive) required to accommodate all of the loop variant values without using spill code. It is assumed that *the lifetime of a value starts when the producer is issued and ends a cycle before the execution of the last consumer begins.* Because a lifetime value may span more than II cycles, successive productions can overwrite previous ones before their last use, thus they must be allocated to distinct storage locations. The model assumes that once a value is stored in a given register, it remains there until its last use. The value of MaxLive is used to compare register requirements when considering a conventional RF against the requirements of a queue organization.

- *Register Usage - Conservative QRF Organization:* As already said, queue structures may be considered to organize the access to register storage locations. In the first stage of this investigation it was assumed that only values having the *same lifetime* can be stored in the same queue. A further constraint imposed was that those values cannot be produced at the same cycle, i.e., two values cannot be written to the same queue at the same time. Accordingly it is calculated the *number of distinct queues* necessary to accommodate all the values produced and used by a modulo scheduled loop. It is also calculated, for each one of these queues, the maximum number of live values at any time, which determines its *longest length* and consequently the minimum size it should be. The total number of queues required and their maximum length can be used to estimate the total number of *queue positions* necessary to accommodate all the values produced during the code execution.

- *Register Usage - Aggressive QRF Organization:* Further investigations showed that it is feasible to relax the conservative condition described above, allowing queues to be shared by register values with distinct lifetimes as shown in Section 3. That condition was used to estimate the number of queues, storage positions and other related measures for a modulo schedule loop assuming this kind of register file organization.

4.3 Machine Models

We have considered three basic machine models in our experiments, whose main specifications are given in Table 2. All of the FUs are assumed as being fully pipelined, allowing any operation to start at any cycle.

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
Total issue width			4	6	12

Table 2: Functional units for three target machine configurations

4.4 Benchmark

To evaluate the effectiveness of queues as an alternative to conventional registers all eligible innermost loops from the Perfect Club Benchmark were scheduled using the framework described above. A total of 1258 loops suitable for software pipelining were used, representing applications in a number of areas of science and engineering. Only loops without subroutine calls and without conditional exits were selected. Loops with conditional structures in their bodies have been IF-converted, which allows each one of them to be viewed as a single basic block. The optimizations and the data dependence analysis were performed by the ICTINEO compiler [3], which supplies only the relevant information that is used by the experimental framework as input data set. This information consists of a set of machine operations (e.g. load, stores, add, mul), and the data and control dependences existing among them. The input set also contains information about loop invariant values, though in this study such values are ignored. The loops and related data and control dependence information were provided by Josep Llosa [15].

4.5 Results

The main objective of this preliminary evaluation was to gain insight into the machine resources required by the new queue register file organization. Two sets of results were produced in this first experimental evaluation, one referring to the conservative QRF and the other referring to the aggressive QRF. The next subsections present some of the results obtained when considering the aggressive organization, along with a few remarks and conclusions which may guide future developments. It should be noticed that any reference to register requirements, number of queue positions and related measures always refers to the value of MaxLive for loop variants, i.e. the maximum number of live registers at any cycle in the schedule.

4.5.1 Number of Queues Required

Any realistic machine design must take into account that resources are finite in number, and certainly this is the case if one is employing queue structures to organize a register file. Ideally the number of queues should be enough to store all the data values produced without the need of spill code. In practical terms this is unlikely to be always possible, but a good design should meet this criteria in most of the cases. The graphics presented in Figure 8 shows the fraction of loops, from the set of 1258 loops considered, that can be scheduled employing only a maximum given number of queues. The diagram shows the results for the 3 machine configurations described above, and the notation "Queue RF - Multiple LF/Q" means that an aggressive QRF was employed. The results suggest that it is possible to schedule more than 95% of all the loops, for any of the machines, with no more than 24 queues, which can be viewed as the size of the name space required. It can also be seen that with a fixed number of 32 queues it is possible to schedule most of the loops regardless the number of functional units, which suggests the model scalability is not constrained by this resource.

4.5.2 Number of Queues Required by Distinct QRF Organizations

As established in Section 4.2 two queue organizations have been considered in this work, one allowing only a single lifetime value to be stored in a queue and the other allowing multiple lifetime values in the same queue. The data presented in Figure 9 shows the total number of queues required when considering both of the strategies. It can be seen that for 70% of the loops there is no significant difference between the two options, but for the other 30% fraction there is in fact an advantage in using the more aggressive organization. The effectiveness of the aggressive organization is particularly evident for the last 10% fraction of loops, which are in fact the most important one when comparing distinct techniques and machine organizations as those loops are the most demanding ones in terms of machine resources.

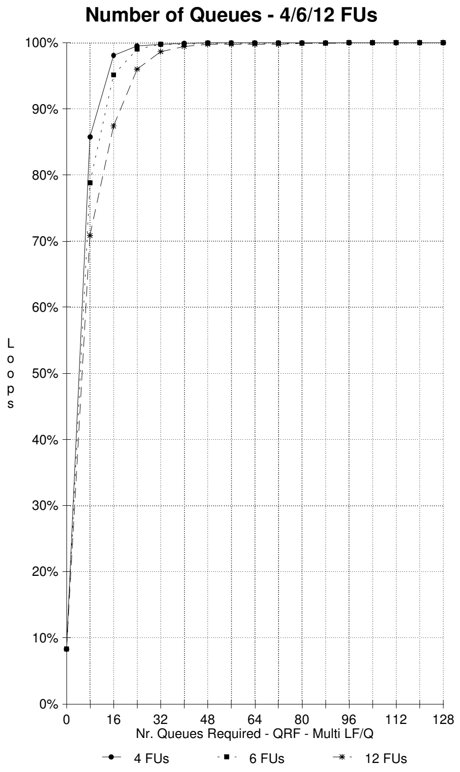


Figure 8: Number of Queues Required

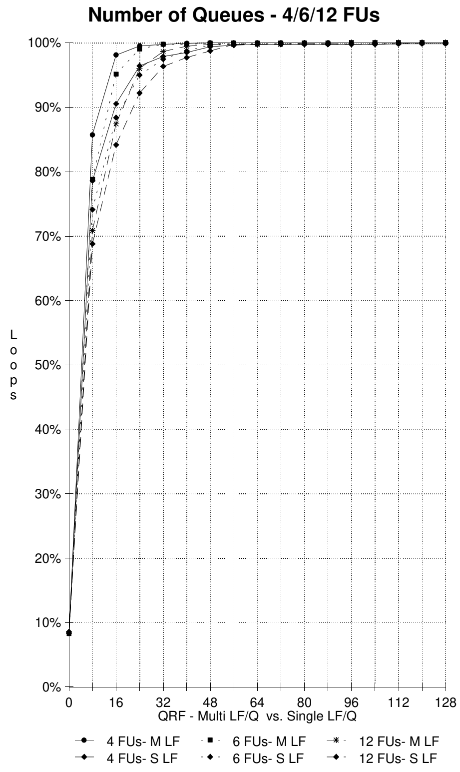


Figure 9: Comparison Between Queue Organization Strategies

4.5.3 Number of Storage Positions Required

In Figure 10 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 at least 90% of all the loops using no more than 48 queue positions, depending on the machine configuration. 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 research groups [18, 8, 10] found that it is possible to schedule around 90% of all the loops with 32 registers, which may suggested that their schemes are more efficient regarding this aspect. A more precise comparative analysis seems to be difficult to perform at this stage as the machine models employed by those authors differ from the ones used in this work. The data presented in Figures 11, 12, and 13 show, for each machine configuration, a comparison between the number of storage positions required when using a QRF and when using a hypothetical RF similar in concept to the ones considered by the cited authors. A significant difference was encountered for a fraction of 20% of the loops, which is due to the fact that in a RF a value is kept in the same register during all its lifetime, while in a QRF the same value may coexist in distinct queues if it is to be consumed at distinct cycles. We are currently working in alternatives to reduce the number of storage positions required, as can be seen in Subsection 4.5.6. Even though some of the results show that our model requires a larger number of storage positions than conventional schemes, we believe that it still has an advantage in terms of silicon area, which is a more critical parameter than the absolute number of physical locations. Once again the figures suggest that the queue organization may allow the machine to scales up without a dramatic increase in the register requirements, which has been reported as unlikely to occur when using a conventional RF.

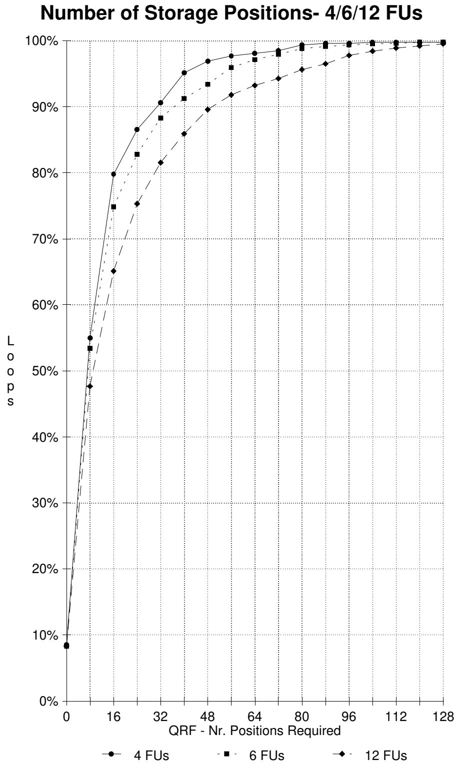


Figure 10: Queue capacity required for 4, 6 and 12 FU models

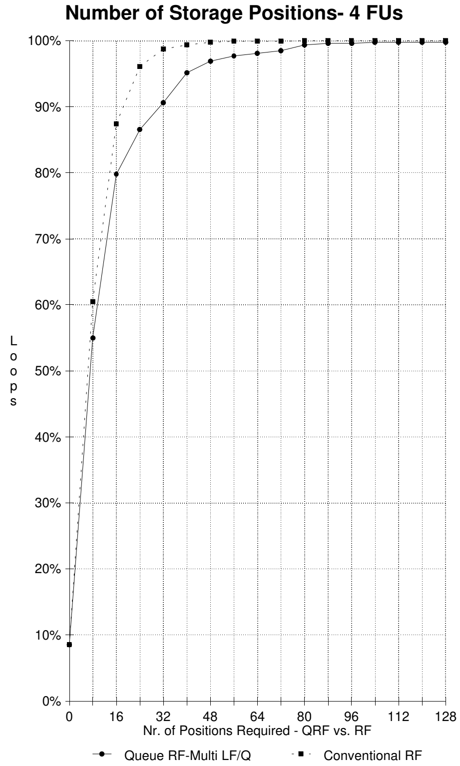


Figure 11: Comparative register requirements – QRF vs. RF with 4 FUs

4.5.4 Loops that Benefit from Greater Parallelism

In order to measure the performance gain in terms of execution time when a machine model A scales up to B, we have informally defined a parameter called $II_{speedup}$, which is

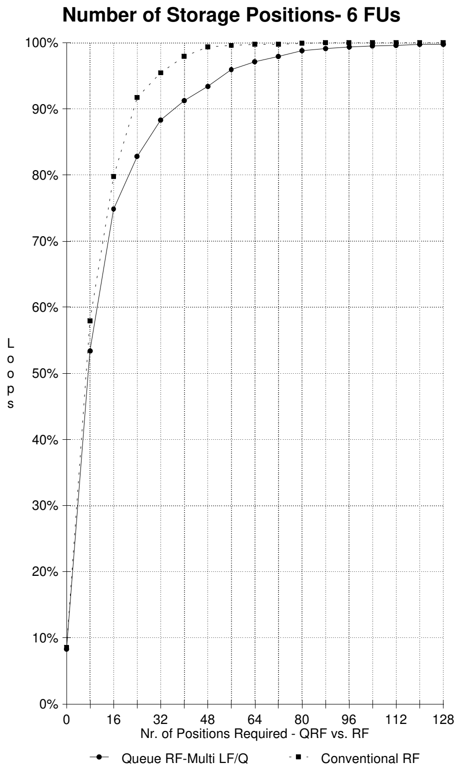


Figure 12: Comparative register requirements – QRF vs. RF with 6 FUs

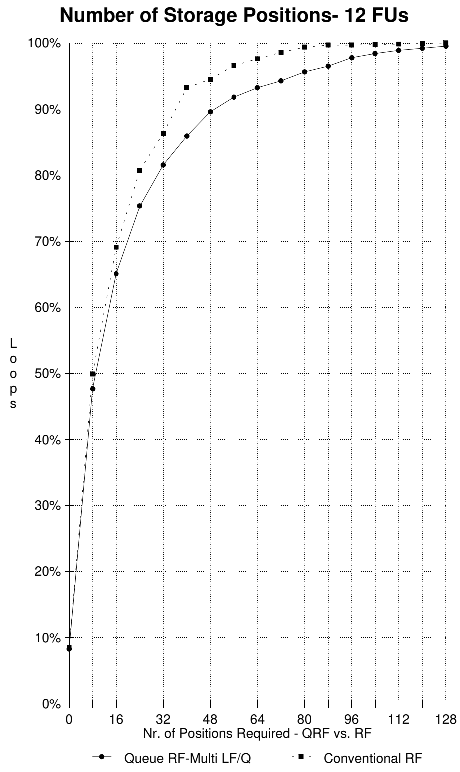


Figure 13: Comparative register requirements – QRF vs. RF with 12 FUs

calculated through this relation:

$$II_{speedup} = \frac{II_{machineA}}{II_{machineB}} \tag{33}$$

Assuming this parameter as valid for this purpose, Figures 14 and 15 shows the number of loops that have their execution time improved when a machine with wider instruction issue is used. We show data referring to 3 possibilities:

- scaling up from 4 to 6 FUs (maximum speedup = 2)
- scaling up from 6 to 12 FUs (maximum speedup = 2)
- scaling up from 4 to 12 FUs (maximum speedup = 4)

Although a simple, this analysis indicates that significant speedups can be attained for a considerable fraction of the loops, suggesting that the use of more aggressive hardware configurations is worthwhile.

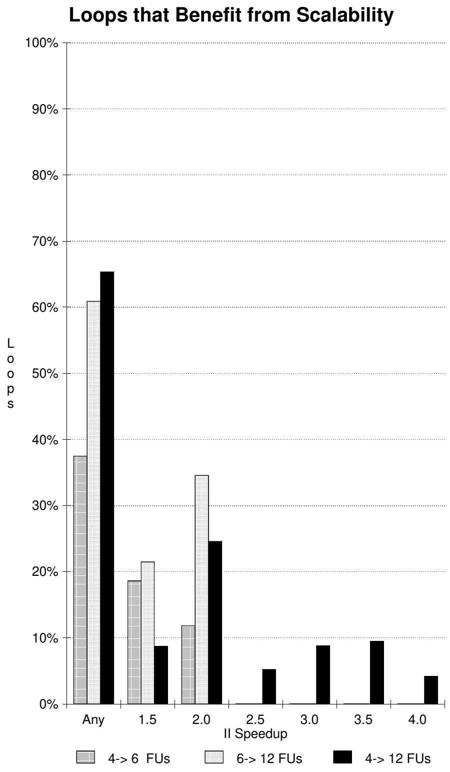


Figure 14: II Speedup

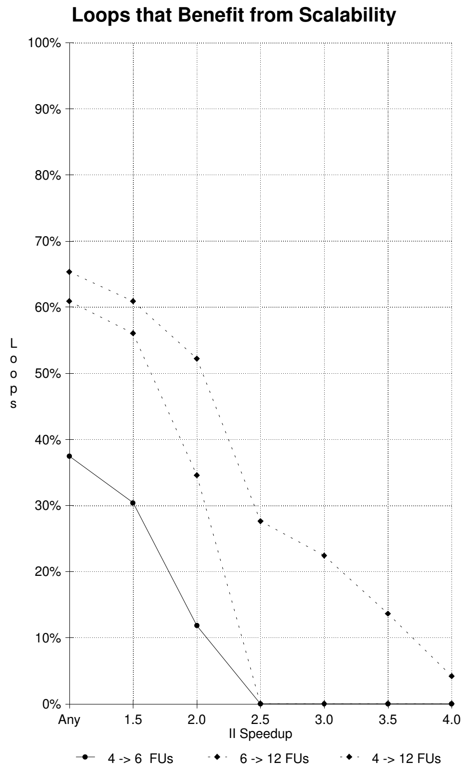


Figure 15: II Speedup – Cumulative Distribution

4.5.5 Trade-Off Between Execution Time and Machine Resources

The data in figures 16 and 17 show the "price" to be paid, in terms of an increase in the number of queues required, to obtain a given $II_{speedup}$ in a more sophisticated machine. It should be noticed that although this graphic representation is not very accurate as many of the points are coincident, it is still able to evidence a pattern in the data distribution. The data on Figure 16 shows the $II_{speedup}$ achieved when the machine model scales up from 4 to 6 FUs. The loops that are resource constrained by ADD and MUL operation were able to benefit from this more aggressive configuration, achieving in some cases the best possible improvement ($II_{Speedup} = 2$). The diagram also shows the corresponding increase (or decrease in a few cases) in the number of queues required. In most of the cases the number of extra queues required is less than 5, and never bigger than 10. In a few cases there is a decrease in the number of queues required, which can be explained by an increase in the size of each individual queue. Similarly the data presented in Figure 17 shows the same analysis performed when a 4 FUs machine is scaled up to 12 FUs. It can be seen a relatively even distribution of the $II_{speedup}$ over all the possible range of improvement (between 1 and 4), which is due to the diversity of machine resources required by each loop. In most of the cases the number of extra queues required is less than 10, although some others required up to 15 extra queues. A few cases required more than 15 extra queues, and again it was surprisingly found that some of the loops achieved a significant $II_{speedup}$ and required less queues for that.

Similarly the data presented in Figure 18 show the number of extra storage positions required when a 4 FUs machine is scaled up to 6 FUs. It can be seen that in most of the cases the number of extra positions required is less than 20, and less than 40 when the machine scales up from 4 to 12 FUs (Figure 19). Those results suggest that the overhead in terms of number of queues may be acceptable when the number of functional units is increased, however we believe that the number of extra storage positions required is too high in some cases, which we hope to improve with further research.

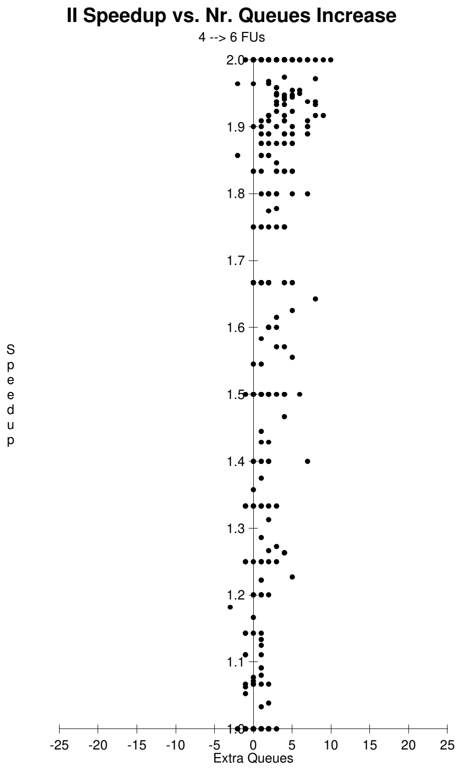


Figure 16: Speedup vs. increase in nr. queues ($4 \rightarrow 6$ FUs)

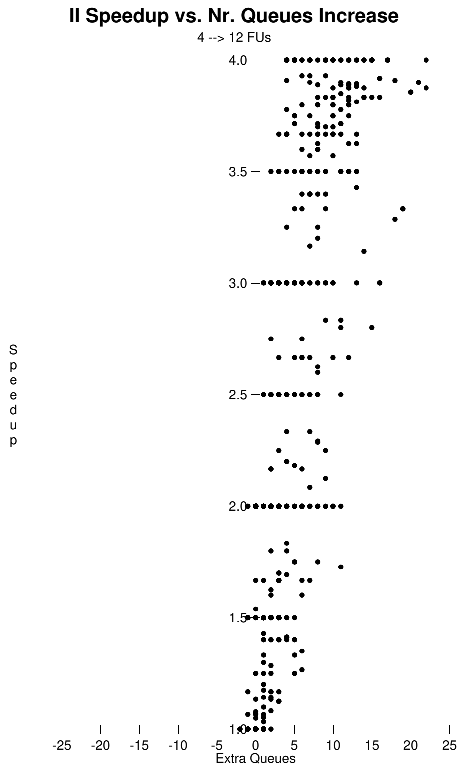
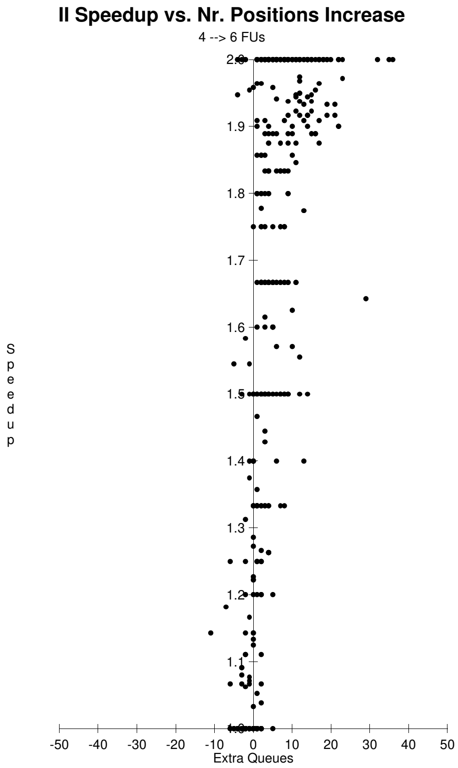


Figure 17: Speedup vs. increase in nr. queues ($4 \rightarrow 12$ FUs)



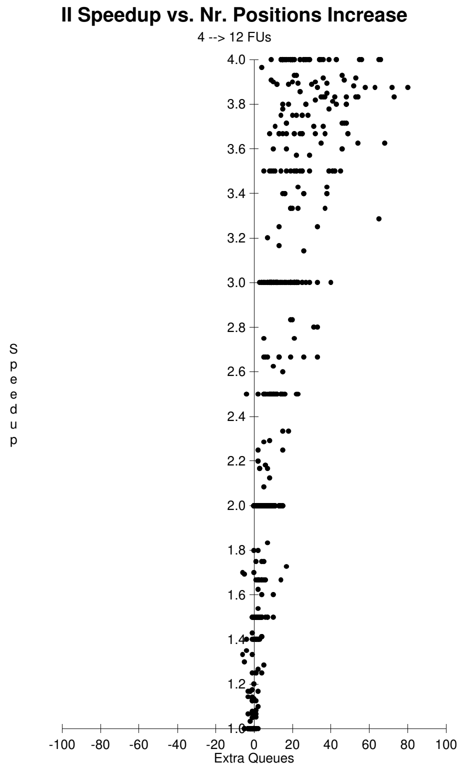


Figure 18: Speedup vs. increase in queue register requirements (4 $\rightarrow$ 6 FUs)

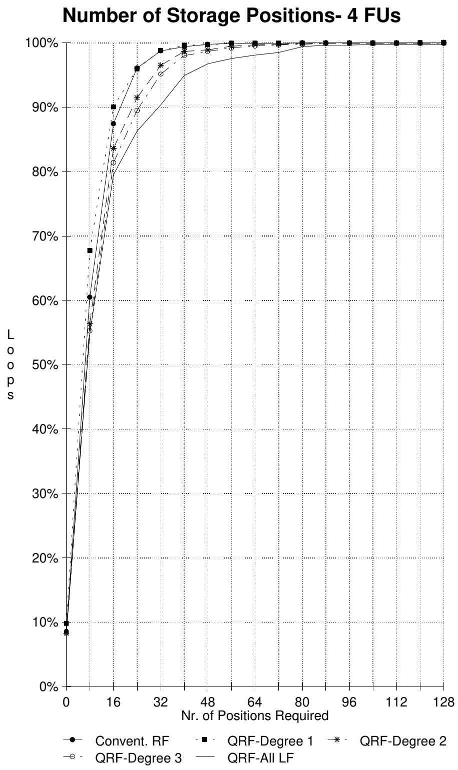
Figure 19: Speedup vs. increase in queue register requirements (4 $\rightarrow$ 12 FUs)

4.5.6 Reducing Pressure on the Number of Storage Positions

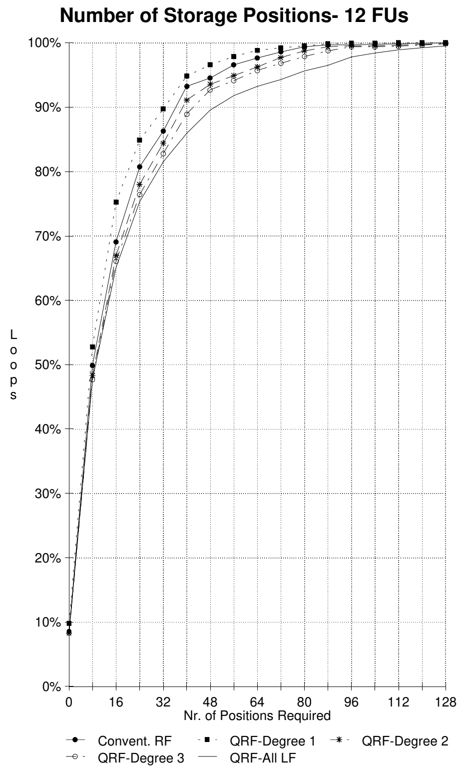
The conclusion we have drawn from the results obtained so far is that the number of actual storage positions required by a QRF is higher than a conventional shared register file organizations. One option to reduce the differential is to keep some of the most heavily used values in a small conventional RF. The idea is that if an operation produces a value that is used several times (a node with many edges), this value will be live at distinct queues at the same time. If this value could be stored in an auxiliary register file (ARF) than a number of queue storage positions could be saved using a small number of registers in the ARF. Figures 20 (a) and 20 (b) show the number of storage positions required under a number of distinct strategies to distribute lifetime values between the QRF and the ARF, as described bellow:

- *RF* - A conventional multiported RF storing all lifetime values.
- *QRF - All lifetimes:* A QRF storing all lifetime values.

- *QRF - Degree 3:* A QRF storing lifetime values that are consumed at most three times (they are produced by a node of degree 1, 2, or 3 in the loop data dependence graph). The remaining data (those produced by a node of degree greater than 3) should be stored in the ARF.
- *QRF - Degree 2:* A QRF storing lifetime values that are consumed at most twice (they are produced by a node of degree 1 or 2 in the loop data dependence graph). The remaining data (those produced by a node of degree greater than 2) should be stored in the ARF.
- *QRF - Degree 1:* A QRF storing lifetime values that are consumed only once (they are produced by a node of degree 1 in the loop data dependence graph). The remaining data (those produced by a node of degree greater than 1) should be stored in the ARF.



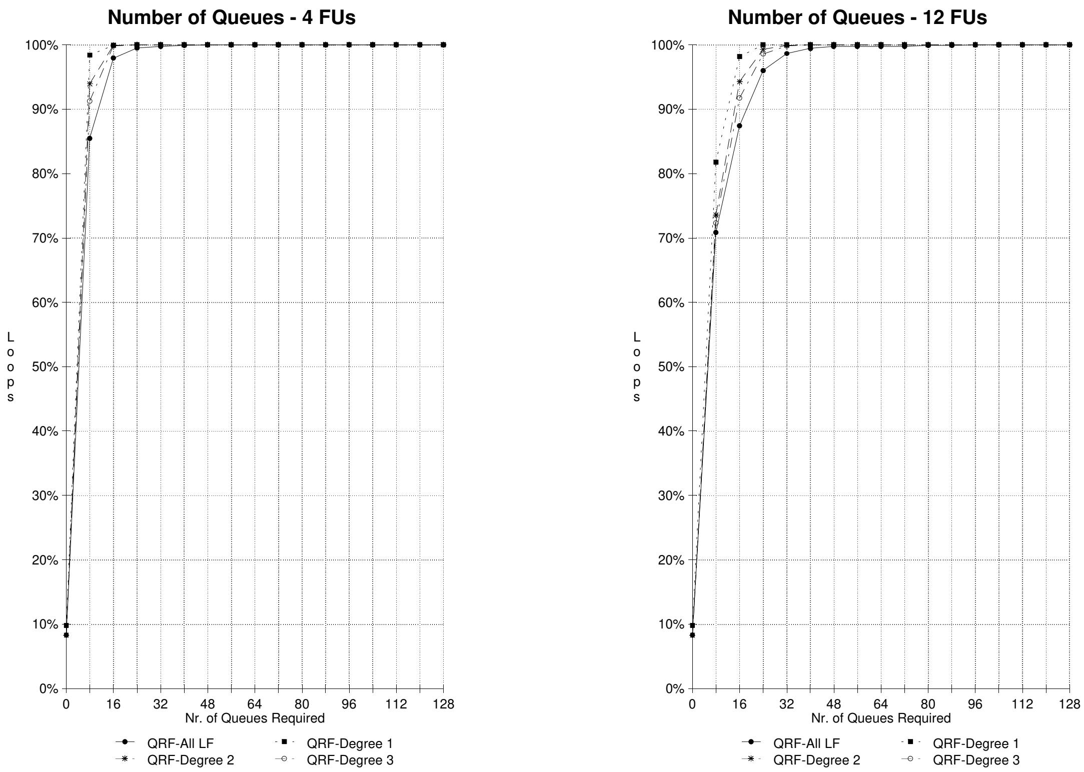
(a). four functional units



(b). twelve functional units

Figure 20: Total queue register requirement when auxiliary RF is used.

As expected the pressure on the number of queue storage positions decreases as the strategy adopted moves from an unrestricted allocation of lifetimes to queues (all lifetime values) to the most restrictive one (QRF-degree 1). In this last case the number of positions required is even smaller than for a conventional RF, particularly for the 12 FUs machine. Figures 21 (a) and 21 (b) show that the number of distinct queues required also decreases when an ARF is used.



(a). four functional units (b). twelve functional units

Figure 21: Number of queues required when auxiliary RF is used.

A key factor to consider the use of this option is the size of the ARF, which will be used to store all of the other values that will not be stored in the queues. Figure 22 shows the size of such ARF according to the type of lifetimes that are stored in the QRF (QRF degree). It should be noticed that the size of the ARF is independent on the number of functional units as the values that will be stored depend only on the topology of the loop data dependence graph. It can be seen that an auxiliary RF of size 16 is able to support the schedule of more than 95% of the loops, and a RF of size 32 could support almost all of them.

A general conclusion drawn from this first experiments using an ARF is that the number of registers (of any type) can be minimized using a QRF of degree 1. Given this alternative we estimate that to schedule almost all of the loops from our input data set requires the

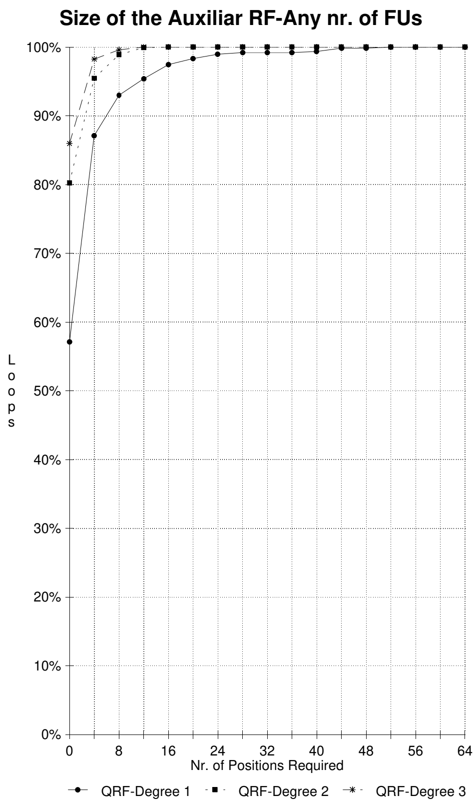


Figure 22: Size of the Auxiliary Register File

following minimum machine resources:

- *Machine A - 4 FUs*: 16 queues, having a total of 40 positions, and an auxiliary RF of size 32.
- *Machine A - 6 FUs*: 16 queues, having a total of 56 positions, and an auxiliary RF of size 32.
- *Machine A - 12 FUs*: 24 queues having a total of 64 positions, and an auxiliary RF of size 32.

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 an experimental evaluation using 1258 innermost loops from the Perfect Club Benchmark.

We started working assuming a conservative condition under which two lifetime values can share a queue, and then we found and proved a new condition under which lifetimes having distinct values can share the same storage queue. According to experimental results adopting this condition implies in a smaller number of machine resources required, thus it has replaced our first conservative assumption.

Considering the early stage of this research work, we understand that the number of distinct queues required by most of the loops is acceptable, which automatically address the name space problem. Although not high for a large fraction of the loops, the total number of storage positions required is a matter of concern in some cases, and a first attempt to improve that was made by using an auxiliary register file. The results obtained are promising, but further investigations are required to validate this alternative, particularly regarding the hardware complexity involved in its implementation. We are currently working in the development of heuristics to reduce the number of queues and storage positions required. In spite of those results we believe that our model still has an advantage over existing ones in terms of the silicon area required to implement the register file.

We have also investigated the potential for scalability of the model being proposed. A considerable fraction of the loops from our input data set would effectively benefit from more aggressive machine configurations, which in turn does not seem to require an unacceptable amount of extra queues and storage positions. That suggests that our model compares favorably in this matter against other VLIW machines proposed in the literature.

We should extend the work done so far by defining a machine model organized by means of clusters composed of functional units and a private register file, which in turn may be organized in a conventional way or by means of queues. The possibility of implementing the communication between clusters by means of queues will also be considered, and new code partitioning and scheduling techniques should be developed.

References

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[2] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th Annual Symposium on Principles of Programming Languages*, January 1983.

[3] 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.

[4] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, pages 345–420, December 1994.

[5] 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*, December 1992.

[6] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.

[7] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register pressure requirements of a modulo schedule. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, November 1995.

[8] 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.

[9] R. Govindarajan, E. Altman, and G. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the MICRO-27 - The 27th Annual International Symposium on Microarchitecture*, November 1994.

[10] R. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the SIGPLAN'93 - Conference on Programming Language Design and Implementation*, 1993.

[11] 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.

[12] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Inc., Englewood Cliffs, N. Jersey, 1991.

[13] R. Jolly. A 9-ns, 1.4-gigabyte/s, 17-ported CMOS register file. *IEEE Journal of Solid State Circuits*, pages 1407–1412, October 1991.

[14] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 - Conference on Programming Language Design and Implementation*, 1988.

[15] J. Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops*. PhD thesis, UPC - Universitat Politecnica de Catalunya, January 1996.

[16] J. Llosa, A. Gonzalez, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT'96*, October 1996.

[17] J. Llosa, M. Valero, and E. Ayguadé. Heuristics for register-constrained software pipelining. In *Proc. of the 29th Annual Int. Symp. on Microarchitecture (MICRO-29)*, pages 250–261, December 1996.

[18] 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.

[19] W. Mangione-Smith, S. Abraham, and E. Davidson. Register requirements of pipelined processors. In *Int. Conference on Supercomputing*, pages 260–246, July 1992.

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

[21] 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.

[22] 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.

[23] 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.

[24] R. Rau and J. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, pages 9–50, May 1993.

[25] D. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[26] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Pres, New York, 1991.