Computer Systems Group

Extending a VLIW Architecture Model

by

Marcio Merino Fernandes and Josep Llosa and Nigel Topham

CSG Report Series **ECS–CSG–34–97**

Computer Systems Group **December 1997**

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

Extending a VLIW Architecture Model

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

Technical Report ECS–CSG–34–97
Department of Computer Science
University of Edinburgh

December 4, 1997

Abstract: This technical report presents further developments on a VLIW architecture model previously reported. A distinctive characteristic of the 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 simple partitioning algorithm has been implemented to perform some experiments with a clustered architecture, an organization we understand as being essential to design and implement a wide-issue machine. We also report a preliminary discussion on some hardware options to implement a queue register file.

Keywords: Instruction-Level Parallelism, VLIW, Software Pipelining, Modulo Scheduling

[*]University of Edinburgh, Department of Computer Science, e-mail: mmf@dcs.ed.ac.uk
[†]Universitat Politècnica de Catalunya, Dep. d'Arquitectura de Computadors, email: josepll@ac.upc.es
[‡]University of Edinburgh, Department of Computer Science, e-mail: npt@dcs.ed.ac.uk

1 Introduction

Current computer architecture technology relies on two basic approaches to improve microprocessor performance: higher clock rates, resulting in more operations being executed in an given period of time, and instruction-level parallelism (ILP), a set of hardware and software techniques that allows a number of machine operations to execute in parallel [29]. A Very Long Instruction Word (VLIW) processor [13] exploits ILP without the need of 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 of parallelism exploitation. Sophisticated compiling techniques have been developed in order to identify parallelism and schedule operations for ILP machines, some of them specifically targeted to optimize the execution of loops, as in many cases they account for the largest fraction in the total execution time of a program. Software pipelining [8] is a technique that allows the initiation of successive loop iterations before prior ones have completed. One class of software pipelining algorithms is modulo scheduling [26], 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 per functional unit in each cycle [6], thus requiring complex hardware organizations with a possible increase in access time. Furthermore, most software pipelining schemes assumes 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 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 [22]. Early designs proposed alternative register file organizations to deal with the problem. The Polycyclic architecture [26] 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 [28] relies on a large number of registers and provides a mechanism to perform a sort of register renaming, 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. Highly multi-ported register files (RF) may not be a good long term solution if FUs scalability is a goal, as we show in [12]. An architecture model comprising clusters of FUs and small private register files seems to be a more reasonable choice as shown by some research works [6, 17]. 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.

We are currently working on 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 [11]. Register files organized by means of queues are believed to be less complex than conventional organizations [2, 14, 15]. That should be the case as the access to a queue is possible only through its head (read operations) and tail (write operations), eliminating the need of address decoding logic to access intermediate positions. On the other hand, this simplification in hardware 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 [12] to decide if values produced by distinct operations 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 $L_a - L_b < (S_b - S_a) \bmod II$.

A number of factors make us believe that a register file organized by means of queues may be an interesting option to be used both private to a cluster and as an inter-cluster communication device:

- *Hardware complexity and silicon area:* As already said, the simpler address decoder employed by a QRF reduces hardware complexity and silicon area. Furthermore, it should be easier to include extra storage locations to an existing queue than to a conventional register file, which is desirable in terms of scalability.

- *Name Space:* In our VLIW machine 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 a QRF may reduce dramatically the pressure on the name space, as shown in [12].

- *Register Allocation:* Register allocation for modulo scheduled loops using a RF imposes further complications due to techniques like modulo variable expansion [27, 19]. We have developed a strategy to allocate data values to queues that eliminates some of the complicating factors presented by those schemes.

- *Code Generation:* Kernel-Only code is a scheme that prevents code size explosion, a side effect often associated with modulo schedule. The use of a queue register file along with support for predicate execution would allow the implementation of this scheme [27].

- *Inter-Cluster Communication:* The efficiency of the inter-cluster communication system is a major and non-trivial issue in a clustered architecture. We are developing a scheme that use queues to implement a sort of asynchronous communication between clusters. The basic idea is to allocate data values to a communication queue as if it was part of the private register file, eliminating the need of extra instructions to move data values between directly linked clusters.

The objective of this technical report is to present and discuss further improvements added to the architecture model proposed in the first stage of this research work [12, 11], as described in the next sections.

2 Research Record

This section presents an overview of previous activities carried out by this research work, which should help to justify and establish the context of the latest developments reported in this document.

2.1 Previous Stage

The motivation for this work is the fact that conventional register files may not be able to support the high register pressure generated by modulo scheduled loops being executed in highly parallel machines. Thus we started working in alternative organizations using queues, which in turn requires new scheduling and register allocation techniques.

The first issue to be addressed was the feasibility of modulo schedule a loop for a VLIW machine assuming the use of a QRF. That showed to be possible through experiments performed using a simple machine model and an implementation of Rau's IMS algorithm [24].

In spite of this, such scheme would be valid only if the number of machine resources required lay within a reasonable limit, demanding a more complete set of experiments to draw a conclusion. The register allocation policy assuming the use of a QRF is one of the the dominant factors to influence the results, so it was carefully handled. The first approach to the problem assumed that only lifetimes having the same lifetime length could share a queue. Then this condition was relaxed, and we showed that under certain circumstances distinct length lifetimes can share the same storage queue, which can be verified by means of the compatibility test presented in Section 1. Experimental results showed it as being a major improvement in terms of resource usage, so that policy has been adopted as a standard procedure in our experiments.

Those preliminary analysis suggested some interesting possibilities, however a more complete set of results was needed to support our conclusions. To accomplish that, the experimental framework was extend to support three distinct machine configurations, comprising a single cluster with 4, 6, and 12 functional units, respectively. Then a total number of 1258 loops from the Perfect Club Benchmark were scheduled for each of those machine models, generating statistical data such as number of queues, number of queue positions, maximum size of each queue, and speedup, among others [12]. Those results showed that considerable performance gains are attainable for a large fraction of loops when the number of functional units is increased. On the other hand the machine resources required when a QRF is used are acceptable, also indicating a clear advantage in terms of scalability when comparing to a conventional register file.

Those conclusions allow us to proceed with this approach, however the simple abstraction level used in the first model means that some problems must be tackled to presented it as a concrete alternative. We are currently working in some of those issues, as we show in the following sections of this report.

2.2 Recent Developments

2.2.1 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 ddg (data dependence graph) in Figure 1a, 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 (Figure 1b). 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, a situation which we shall call *simultaneous writes* (Figure 1c), resulting in at least two complicating factors:

- The instruction format should allow a single instruction to specify an unbounded and possibly large number of destination queues.
- The register file should allow simultaneous access to an unbounded number of queues to perform the write operations specified by an operation.

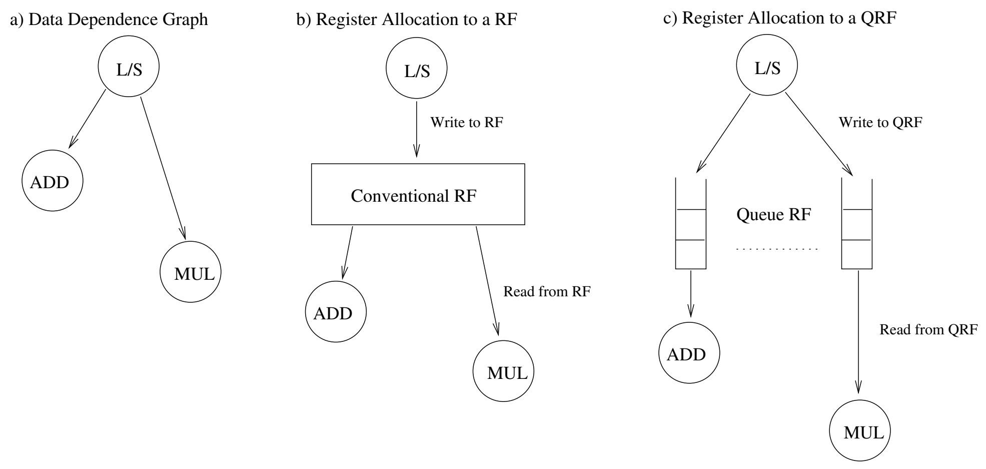


Figure 1: Register Allocation Data-Flow

These two factors alone justify the need for some actions to bring the architecture model to a more realistic level. In this report we propose to modify the ddg by introducing copy operations to eliminate the need of simultaneous writes, as shown in Figure 2. This copy operation has one input and two output operands, being sufficient to eliminate both complicating factors presented above. Further discussions and experimental results using this technique are presented in Section 5.

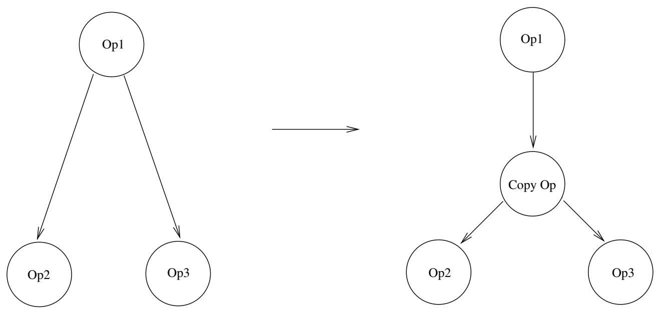


Figure 2: Basic Transformation to Include a Copy Operation

2.2.2 Optimizing FUs usage

In a modulo schedule the number of cycles between the initiation of two consecutive iterations is called Initiation Interval (II), and the minimum value achievable (MII) depends on resource constraints (ResMII) and recurrence constraints (RecMII). The ResMII is calculated according to machine resources available such as number of FUs, pipeline stages, and data path buses. For the sake of simplicity in this work we take into account only the number of individual FUs to calculate ResMII. This is done by dividing the total number of uses required by the most heavily used FU (in a single iteration) by the number of available FUs of that type. It might happen the calculated value of ResMII is not an integer, in which case it would be rounded up to the next integer because the value of II must be an integer. A particular case occurs when the loop does not have enough operations to fully exploit the number of machine resources available, in which case the exact value of ResMII is less than one, and consequently rounded up to one. The process of rounding ResMII up to the next integer causes performance degradation as it allows unnecessary empty slots to exist in the schedule. One way to minimize this problem is to perform *loop unrolling* [9] of the loop body prior to modulo scheduling, an optimization that produces considerable benefits according to Lavery [20]. We have performed a number of experiments with loop unrolling to estimate its effect on our VLIW architecture model, as shown in Section 6.

2.2.3 First approach towards a clustered architecture

We have carried out some experiments showing that significant kernel speedups can be achieved by employing a larger number of FUs. However, the current technology in not capable to produce a high performance single chip containing more than a few FUs, let alone the highly multi-ported register file that such architecture would require [12]. A machine organization consisting of clusters with a few FUs and a small register file seems to be a more reasonable choice, which intuitively favours scalability. Our first approach to the problem is to include some heuristics to the IMS algorithm in order to produce a modulo schedule for a given clustered machine, which means that inter-cluster communication limitations must be taken into account. As discussed in Section 8 we have performed some experiments to evaluate two main issues regarding this approach:

- To estimate how hard are the constraints imposed by a clustered architecture to the modulo scheduler, which can be done by comparing the II values of the schedules targeting single and multi-cluster machines.
- To estimate the machine resources required by a clustered architecture, which would allows us to decide if that is indeed a reasonable strategy.

Further discussions and experimental results about this approach can be seen in Section 8, along with future plans to improve the techniques employed so far.

3 Experimental Framework

This section presents the basic structure of the framework used in our experiments, which consists of a configurable VLIW machine model, a modulo scheduling algorithm, and a register allocator. The basic input to the framework is an *innermost loop*, represented by operations and corresponding data dependence information among them, generating as output a modulo schedule and the machine resources required. The implementation has been done using C++ language and LEDA library routines [23], which are particularly useful for graph manipulation. It should be noted that some new features have been incorporated to the framework, in addition to the ones previously described in [12].

3.1 Machine Model

The target machine model being scheduled for can be viewed as a collection of basic building blocks, which provides enough flexibility to model distinct hardware configurations.

3.1.1 Basic Building Blocks

- *Functional Units:* Used to perform the actual computations and memory operations. All of them are assumed as being fully pipelined, being able to start a new operation at any cycle, which will take a fixed number of cycles to complete *(latency)*.

  For the sake of simplicity we consider only four types of FUs:

  - L/S: Executes memory load and store operations, transferring values between the main memory and the register file.

  - ADD: Executes addition, subtraction, conditional branch, compare, and absolute operations, reading and writing operands to the register file.
  - MUL: Executes multiplication, division, square root and modulus operations, reading and writing operands to the register file.
  - Copy: Executes copy operations, supporting the scheme proposed in Section 5.

- *Queue Register File (QRF):* A register file comprising an unbounded number of queues and queue positions, providing enough access ports to perform the required read and write operations by FUs. Each of the queues can be either read or write by any one of the FUs at any cycle, being the scheduler's responsibility to prevent access conflicts.

- *Conventional Register File (RF):* A multi-ported register file comprising an unbounded number of registers, used basically to compare our approach with conventional ones found in the literature.

3.1.2 Single Cluster Machine

A number of functional units connected to either a QRF (Figure 3a) or a RF (Figure 3b), organized in a single cluster.

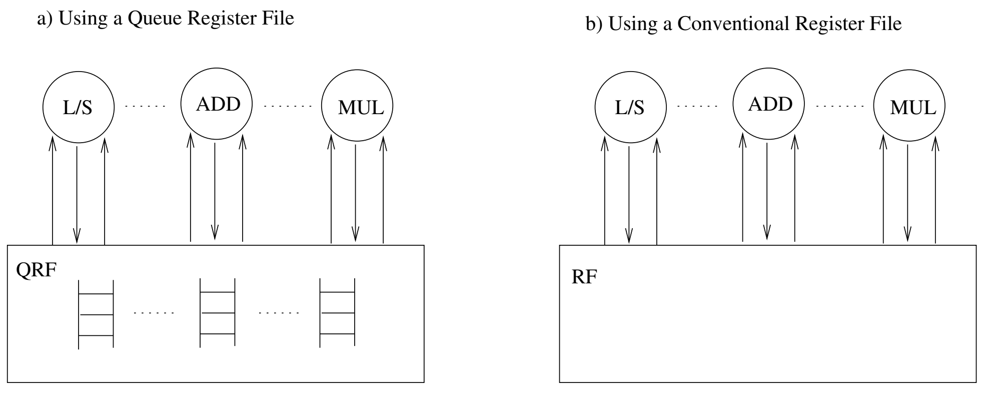

a) Using a Queue Register File
L/S
ADD
MUL
QRF
b) Using a Conventional Register File
L/S
ADD
MUL
RF

Figure 3: Single Cluster Machine Organization

3.1.3 Clustered Machine

A number of single cluster machines, each of them comprising three standard FUs (L/S, ADD, MUL), a Copy FU, and a QRF. The clusters are interconnected by means of a bi-directional communication ring, also implemented by means of QRFs. As an example, Figure 4 shows the building blocks organization of a four-cluster machine.

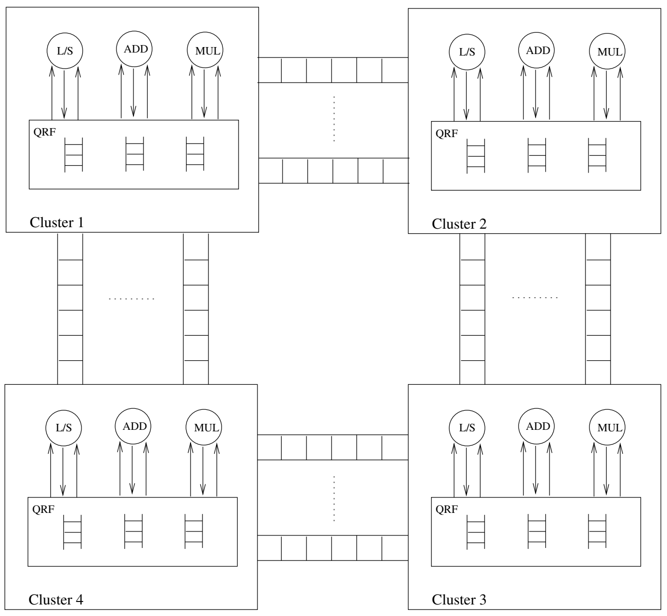


Figure 4: Example: Four-Cluster Machine

3.2 Modulo Scheduling Algorithm

Rau's *Iterative Modulo Scheduling (IMS)* [25] is the basic algorithm used in this framework, which has been very successful in terms of the quality of the schedules produced for our input data set, managing to achieve $II = MII$ in most of the cases. In terms of execution time it may be considered acceptable for most of the loops, i.e. those comprising less than one hundred operations. As the number of operations grows there is a noticeable degradation in performance, which can be explained by the backtracking process employed by the algorithm. However in our input data set that is the case for just a few loops (28 of them, without performing loop unrolling). We understand it as not being a problem as the main objective at this stage is to evaluate the feasibility of this new approach. That would not be the case in a production compiler, however other possibly more efficient algorithms (e.g. Swing Modulo Scheduling [21]) could be used without the need of extensive modifications.

3.3 Register Allocation

So far we have performed register allocation only for *loop variant* values as they account for a much larger fraction of register requirements. We should also perform the allocation of loop invariants in the near future, which will require an alternative scheme due to the QRF characteristics. That should not have a dramatic effect on the results obtained in terms of machine resources, although it is an issue that must be tackled in this new architecture model.

3.3.1 Allocation to a Queue Register File

Register allocation is performed assuming an infinite number of queues and queue positions, however we try to minimize the usage of the most critical resource (number of queues) by allocating as many values as possible to a single queue. As values are allocated to queues instead to individual registers, the number of distinct queues can be seen as the size of the *name space* required. After a modulo schedule is produced an interference matrix is built, providing information about every pair of lifetimes that are able to share the same storage queue. The compatibility test presented in Section 1 is used to create the matrix, which requires information about the *start cycle* and *total length* of every lifetime generated by the schedule. Then a single pass procedure is executed, trying to allocate a lifetime to any of the existing queues, or using a new one if that is not possible due to incompatibilities. The number of queues could be further minimized if a more efficient procedure like *graph coloring* [7] were employed, which we should implement in a future version of this framework.

3.3.2 Allocation to a Conventional Register File

Actual register allocation to a RF is not performed but instead, after a modulo schedule is produced, the maximum number of live values *(MaxLive)* is calculated, a parameter that states the largest number of distinct values that must coexist at a given cycle. As each distinct value must be stored in a distinct location until the last read operation is performed on it, MaxLive can also be seen as the minimum size required by a conventional register file. We have used this parameter to compare with the results obtained when a QRF is used. It is acknowledged that modulo schedules often requires more register names than the available architecture visible registers. This demands alternative techniques to deal with the problem like modulo variable expansion [19], leading to an undesirable code size explosion in some cases. Assuming that no such techniques are used (which may further increase the pressure) MaxLive can also be viewed as the size of the name space required to allocate loop variants when a RF is used.

3.4 Input Data

An innermost loop is the basic unit taken as input by the framework, read in a format that should include all of the loop *operation codes* plus the existing *data dependence information* among them, which will be used by the IMS algorithm to build a data dependence graph. All eligible innermost loops from the Perfect Club Benchmark [5], a total of 1258 loops suitable for software pipelining are used. Only loops without subroutine calls or conditional exits were selected. If-conversion [1] was applied to eliminate conditional structures from their bodies, allowing each of them to be viewed as a single basic block. The optimizations and the data dependence analysis were performed by the ICTINEO compiler [3]. Information regarding loop invariants are also provided, which should be used in the near future when register allocation procedures for them will be developed.

3.5 Output Data

For each loop scheduled a number of output data is generated, which can be used for specific analysis, or most often by means of statistical data regarding the whole set of loops. For each loop scheduled the main output data provided by the experimental framework are:

- Modulo schedule and corresponding parameters such as *MII, II, schedule length, and stage count (SC)*. To ensure correctness a *check procedure* is executed at the end of the schedule generation phase, verifying if all data dependencies and machine resource constraints are properly handled.
- Number of queues required to allocate loop variant values, also viewed as the size of the name space.
- Longest size of a queue refers to the maximum number of live values that must coexists in a queue at a given cycle, which also corresponds to the lower bound on its required storage capacity.
- Number of queue positions, which is calculated by summing up the longest size of every queue, indicating the total number of storage positions required.
- Number of registers required by a conventional register file (MaxLive).
- Scalability analysis is concerned with the changes on II and stage count when more FUs units are used to schedule the same loop. We have defined a parameter called $II_{speedup}$ to measure the gain in performance execution of the *kernel code* when a given machine model A scales up to a machine model B, which is calculated through the following expression:

  $$II_{speedup} = \frac{II_{machineA}}{II_{machineB}} \tag{1}$$

  However the loop execution time is not only influenced by the performance of the kernel phase, but also by the *prologue* and *epilogue* phases, which are directly influenced by the value of the stage count. In our model of execution a larger stage count means that the less efficient prologue and epilogue phases will be longer, thus an increase in the value of the stage count possibly means performance degradation. A parameter called SC_{dif} has been defined in order to account for the stage count variation when distinct machine configurations are used, being calculated through the following expression:

  $$SC_{dif} = SC_{machineB} - SC_{machineA} \tag{2}$$

- Enhancement Costs analysis uses both parameters above defined, $II_{speedup}$ and SC_{dif}, however in another context: to quantify the variation observed on these parameters due to new features added to the experimental framework.

3.6 Influence of II and Stage Count on Execution Time

We have affirmed that a decrease in the value of the II implies in improvement on the kernel code execution time, as well as a smaller stage count implies in less time spent in the slower prologue and epilogue phases. Some of the features added to the architecture model alters the values of II and SC, an effect that is noticeable when including copy operations or performing loop unrolling. The total execution time of an innermost loop depends on the prologue, kernel, and epilogue phases, and also on the number of iterations *(iteration counter)* of the innermost loop and of the loops surrounding it *(execution counter)*. Thus the value of loop counters must be taken into account to reach a final conclusion on performance gains or losses if conflicting variations occurs on those parameters (e.g. a decrease on II and an increase on SC).

Assuming that kernel only code will be generated, we have derived some expressions intended to estimate and compare the execution times of two distinct schedules for a given loop, which could be used to produce a more precise analysis if loop counter values are known. The following notation is used to derive the expressions:

- II: Initiation interval.
- SC: Stage Count.
- ic: Iteration counter of the innermost loop.
- ec: Execution counter of the innermost loop.

3.6.1 Kernel Code Execution Time

The total number of cycles executed by the kernel code of a given schedule, X_k is:

$$X_k = (II \times ec \times ic) - (II \times (SC - 1) \times ec) \tag{3}$$

3.6.2 Prologue and Epilogue Execution Time

The total number of cycles executed by the prologue and epilogue phases of a given schedule, X_p and X_e respectively, are given by the following expressions:

$$X_p = II \times (SC - 1) \times ec \tag{4}$$

$$X_e = II \times (SC - 1) \times ec \tag{5}$$

3.6.3 Total Execution Time of the Innermost Loop

The total execution time for the modulo schedule of a given innermost loop, X_t, comprises the prologue, kernel and epilogue phases. Thus using the above expressions it is possible to propose an expression to estimate X_t:

$$X_t = (II \times ec \times ic) + (II \times (SC - 1) \times ec) \tag{6}$$

4 Experimental Results - Basic Configuration

This section presents some results regarding the basic configuration used in our first experiments [12], which are presented again to be compared with the new features added to the model. The machine configuration comprises a single cluster of 4, 6 or 12 FUs, as seen in Table 1, connected either to a QRF or to a RF. We shall call this configuration Baseline 1, and the main experimental results are as follows:

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 1: Functional Units for Three Machine Configurations

- Number of Queues: As seen in Figure 5 it is possible to schedule most of the loops for the basic configuration with 32 distinct queues, regardless the number of FUs. It is still possible to schedule a large fraction of loops with only 16 queues for the 4 and 6 FUs machines, however that is not ideal for 12 FUs.

- Name Space - QRF vs. RF: Assuming that the available size of the name space is 32, we have found that it is possible to schedule a larger fraction of the loops using a queue register file (QRF) than using a conventional one (RF), as shown in Figure 6. It should be noticed that the differences between distinct machine configurations are smaller when a QRF is used, suggesting that in this case the size of the name space is less sensitive to the number of FUs, which would favour scalability.

- Total Number of Positions: We have found that the total number of storage positions required by a QRF is larger than required by a RF [22, 10, 16], however we are working in a number of alternatives to improve this factor, in spite of our beliefs that the lower hardware complexity of a QRF could compensate that difference.

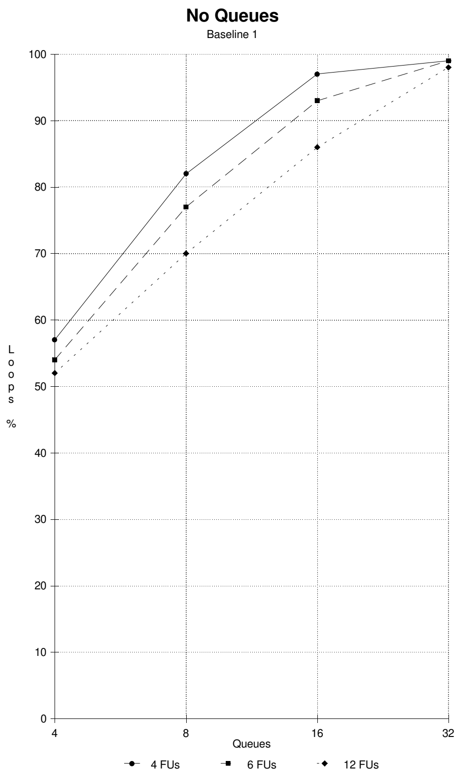


Figure 5: Number of Queues-Baseline 1

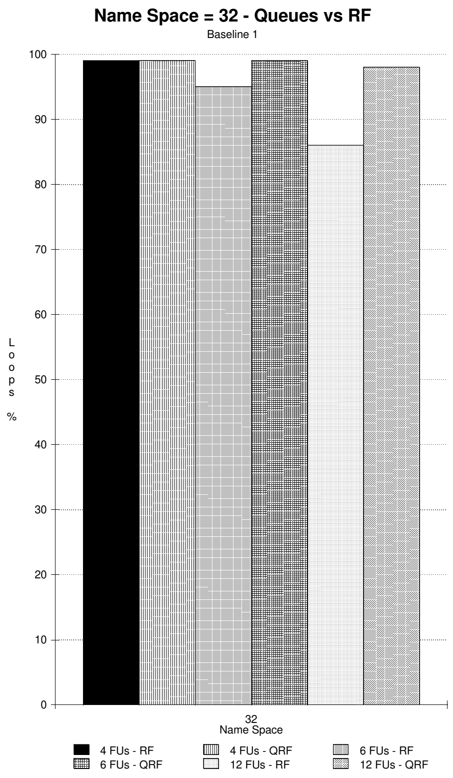


Figure 6: Name Space-Baseline 1

- Scalability - Kernel Speedup: It is possible to achieve reasonable speedup levels for the kernel code execution when the number of FUs scales up from 4 to 6 and 12 respect., as shows the *cumulative distribution* of speedup values in Figure 7.

- Scalability - Stage Count Variation: Increasing the number of FUs implies in an increase of the stage count for some cases, as seen in Figure 8. The variation perceived between 4 and 6 FUs is not large, however it is significant between 4 and 12 FUs, as show the average stage count values (SC_{avg}) in Table 2.

No. Functional Units	Average Stage Count
4	3.7
6	4.2
12	5.5

Table 2: Average Stage Count-Baseline 1

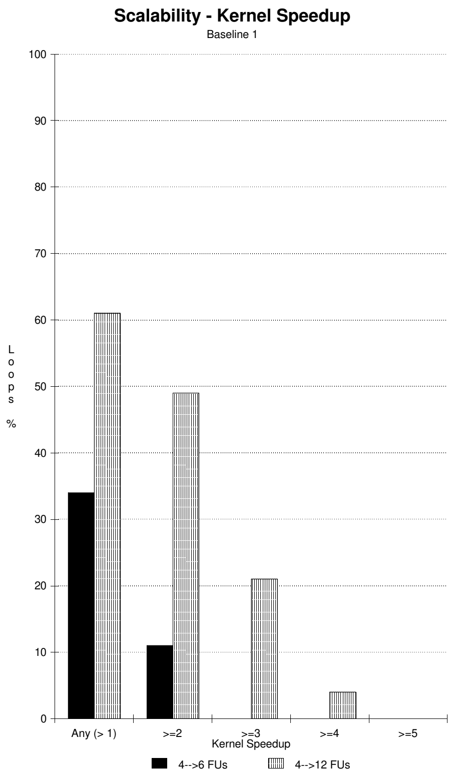


Figure 7: Kernel Speedup-Baseline 1

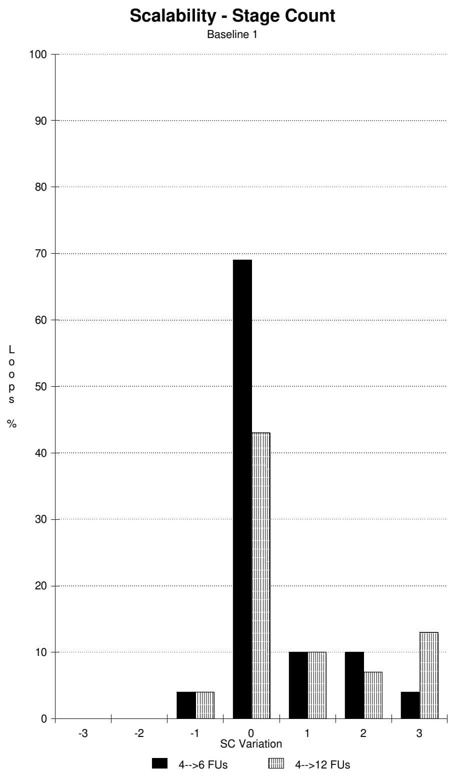


Figure 8: SC Variation-Baseline 1

5 Effect of Copy Operation

As already said in Section 2.2.1 the QRF model adopted implies that a value can not be read more than once, requiring it to be stored in as many queues as the number of times it will be consumed. To deal with the problem we propose the introduction of a copy operation, which should be executed by a dedicated functional unit capable of reading one value from a queue and write it back to two other queues. In terms of hardware cost it requires an extra FU, which should be simple to implement. However it also requires *extra access ports* to the QRF, an overhead that should be further investigated. Extra delays may increase the loop execution time due to alterations to include copy operations in the ddg, so we have performed some experiments and compared the results against the ones obtained for the basic configuration (Section 4).

The machine model has been modified to allow the inclusion of FUs capable to perform copy operations with latency of one cycle, exhibiting the new configuration shown in Table 3. In order to compare this new feature with other machine configurations we have opted *not to take into account copy FUs in the machine's total number of FUs* as we understand them as being supporting FUs, existing only due to the QRF characteristics.

Functional	Operation	Issue	Number of functional units		
unit type	latency	rate	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
copy	*1*	1/˜	*1*	*2*	*4*
Total issue width (without copy Op)			4	6	12

Table 3: New Machine Configurations Including FUs to Execute Copy Operations

The introduction of copy operations is done when the data dependence graph is built, occurring every time there is more than one consumption edge originating from the same node, as seen in the diagram of Figure 9.

It should be noticed that two attributes associated with the modified edges, named *delay and distance*, are also changed in order to preserve the correct data dependence among them. As an example, Figure 10 shows step-by-step the transformations occurred in a ddg when including copy operations to allow four reads on the same value, thus requiring four distinct queues. Special care was taken to design an algorithm able to generate a *balanced* subgraph after the inclusion of copy operations, which should minimizes the overall delay incurred to execute the extra copy operations. The subgraph root is the original productor operation, and the leaves are the original consumers. At the end of the process a checking procedure is executed, comparing the original data dependence graph against the modified one, ensuring that both of them produce the same results.

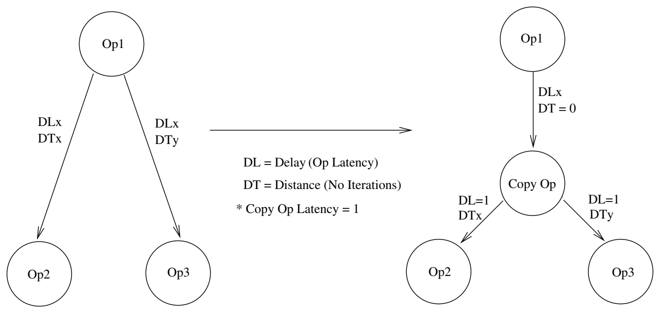


Figure 9: Modifications on the DDG to Include a Copy Op

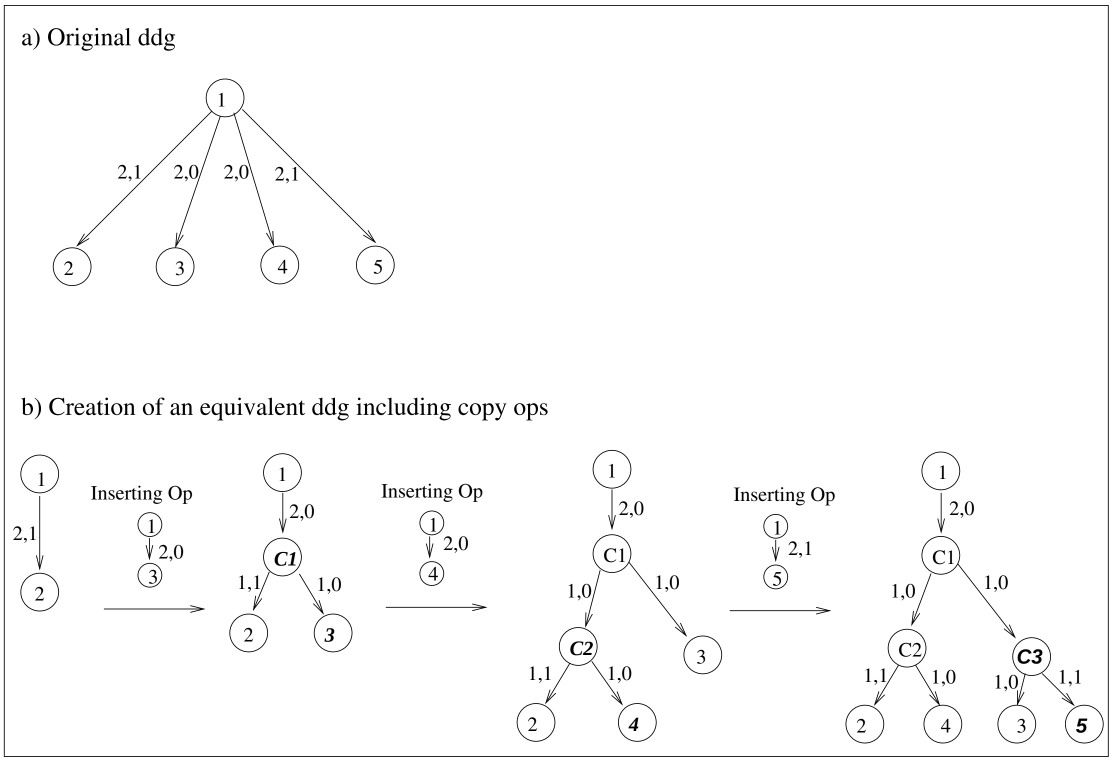


Figure 10: Example: Inserting Copy Ops to a DDG

A number of experiments were carried out to evaluate the effect of using copy operations, some of them are presented as follows:

- Number of Queues: Using copy operations does not increase significantly the number of queues required to schedule a given fraction of the benchmark loops (it even decreases in some cases). That is specially the case for loops requiring 16 or 32 queues, as seen in Figure 11. The main reason for this is that copy operations reduces the lower bound in the number of queues required to store values to be consumed more than once, which compensates the overhead introduced by extra operations in the ddg. As an example it can be seen that the lower bound in the number of queues of the ddg of Figure 10a is 4, however it is only 2 when copy operations are used (Figure 10b).

- Name Space - QRF vs. RF: The introduction of copy operations does not produce any significant alteration in the number of loops scheduled for a name space size of 32, as seen in Figure 12, thus still keeping the advantage of using a QRF instead of a RF. It should be noticed that the data referring to the RF was obtained considering the original ddg, which means that there is no distortions due to unnecessary copy operations.

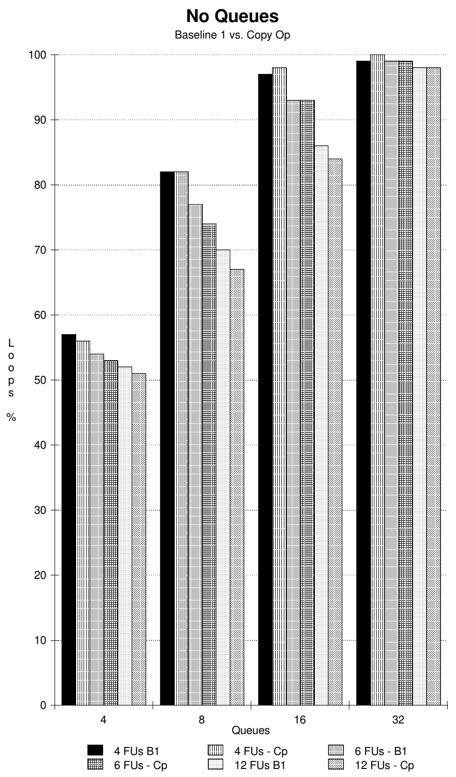


Figure 11: Number of Queues-Copy Op

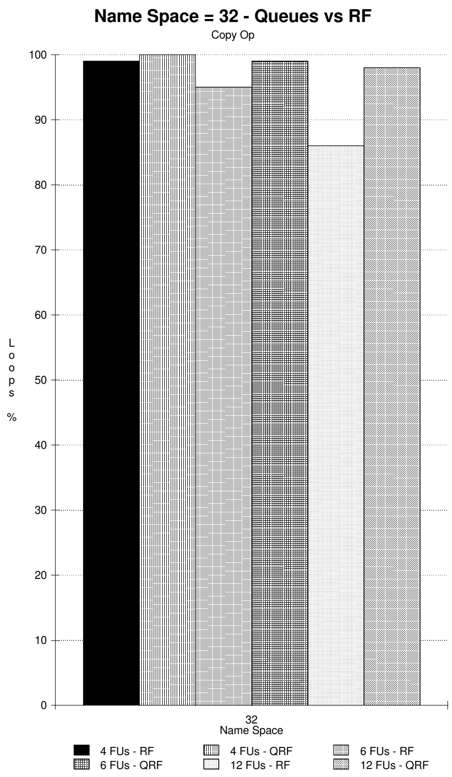


Figure 12: Name Space-Copy Op

- Total Number of Positions: This parameter is closely influenced by the number of queues required, so accordingly copy operations do not change significantly the figures obtained for the basic configuration.

- Initiation Interval Variation: Around 95% of loops were scheduled with the same II after the insertion of copy operations (Figure 13), which means that there is no performance degradation in the kernel execution for most of the cases. For a small fraction of loops the resulting kernel speedup is less than one, indicating an execution delay, however we understand it as not being an intolerable performance penalty as it constitutes only one cycle for most of these cases.

- Stage Count Variation: Inserting copy operations does not change the value of the stage count in about 80% of loops (Figure 14), and when it occurs is typically one extra stage.

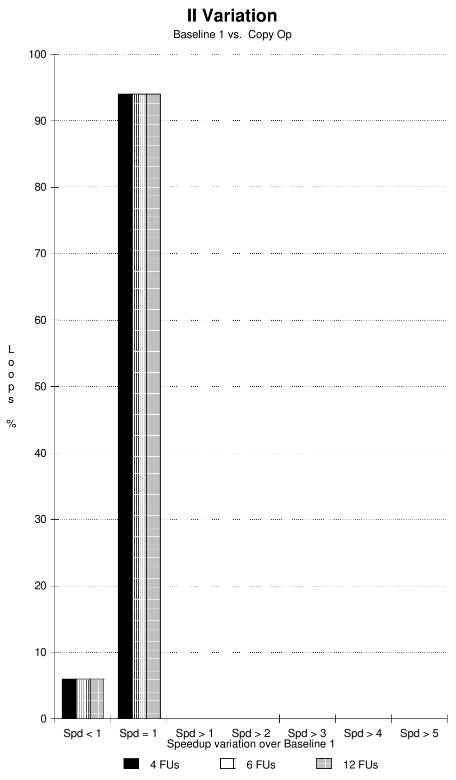


Figure 13: II Variation-Copy Op

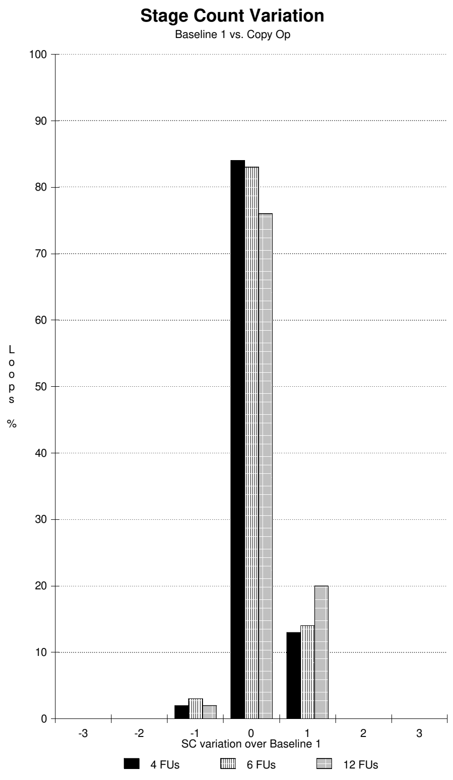


Figure 14: SC Variation-Copy Op

- Conclusions: 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 is a small reduction in the required number of queues and queue positions for the most demanding loops. Apart from the extra access ports required, we understand this as been an acceptable solution to the problem, thus we should seek a way to optimize its hardware implementation.

6 Effect of Loop Unrolling

One of the main objectives of this research work is to investigate a scalable VLIW machine, which implies in a larger number of FUs available for program execution. 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. Loop unrolling [9] is a well known compiler optimization that replicates the body of a loop some number of times called **unroll factor (u)**. This results in a a larger number of operations for the exploitation of the available machine resources. Suppose a given loop has an iteration counter incremented by *step* units. After unrolling by a factor u this counter is incremented by $step \times u$ units, thus reducing the number of iterations executed and consequently the loop overhead [4]. We have used loop unrolling to increase the number of operations to be executed in parallel, and also to optimize the value of II when it is not an integer, implementing an algorithm taken from [30].

As pointed out by several authors unrolling can improve performance, however like other compiler optimizations it can generate side effects that may compromise the benefits achieved. In this work two main issues are of particular concern, and constitute the main points investigated through experimental analysis:

- Unrolling may increase register pressure, requiring in some cases the introduction of spill code, compromising the overall performance.
- Unrolling may generate a loop containing too many operations and complex dependence chains, making it difficult and time consuming for the IMS algorithm to find a valid schedule for a given MII.

To prevent further complications we have opted to perform loop unrolling only if the ResMII is the dominating factor determining the MII, in which case the unrolling factor chosen depends on the performance degradation incurred on rounding ResMII up to an integer. Thus the unroll factor u is chosen if a given tolerance value, called $U_{tolerance}$, is less than a given upper bound [24], which can be calculated by means of the following expression:

$$U_{tolerance} = \frac{\lceil ResMII \times u \rceil}{ResMII \times u} - 1 \qquad (7)$$

If the number of iterations of the original loop is n, the number of iterations performed by its unrolled version is approximately $n \div u$. So, when comparing the kernel code performance of both cases, the *actual II* for the unrolled version should be $II_{unrolled} \div u$, as this is the actual number of cycles required to execute *one* of the original loop iterations.

We present here some experimental data and conclusions regarding the use of loop unrolling in our scheduling framework.

- **Number of Queues**: As expected loop unrolling produces a moderate increase in the number of queues required, although 32 queues are still enough to schedule over

90% of the loops for any of the machine configurations considered (Figure 15). The increase in the number of queues is less noticeable in the case of large loops because in general they do not require unrolling to efficiently exploit machine resources.

- Name Space - QRF vs. RF: Loop unrolling is more demanding in terms of extra register names when a conventional register file is used instead of a queue register file, as can be seen in Figure 16, a difference that may be even more accentuated as more FUs are used, and possibly higher unroll factors.

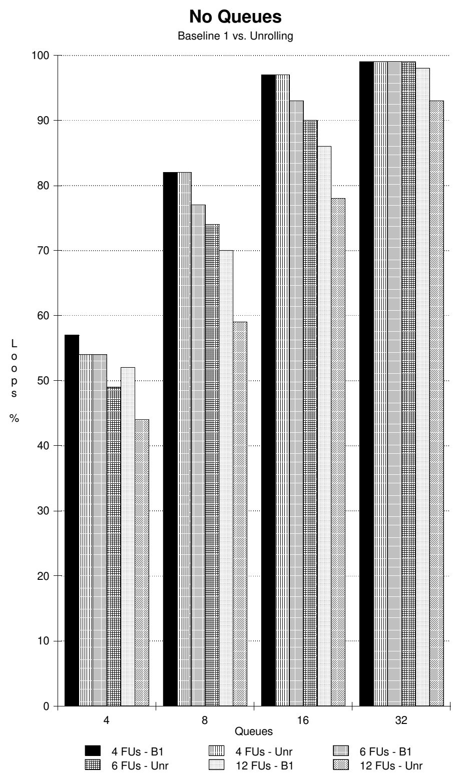


Figure 15: Number of Queues-Unrolling

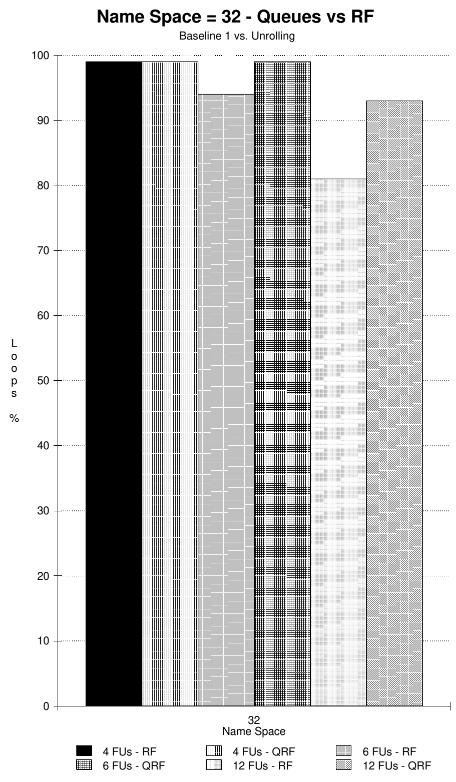


Figure 16: Name Space-Unrolling

- Total Number of Positions: Once again this parameter is influenced by the number of queues required, showing a moderate increase for the less demanding loops.

- Initiation Interval Variation: A considerable fraction of loops achieved some speedup when loop unrolling was applied, which is very significant if it is taken into account that no extra FU was used. The cumulative distribution seen in Figure 17 shows that some performance degradation occurred only for a small fraction of loops. The main cause for this are limitations of the IMS algorithm to schedule more complex loops, an effect that may be minimized if another algorithm is employed.

- Stage Count Variation: Loop Unrolling does not alter the stage count for most of the loops (Figure 18), and when it occurs is often a decrease, which might improve the overall performance.

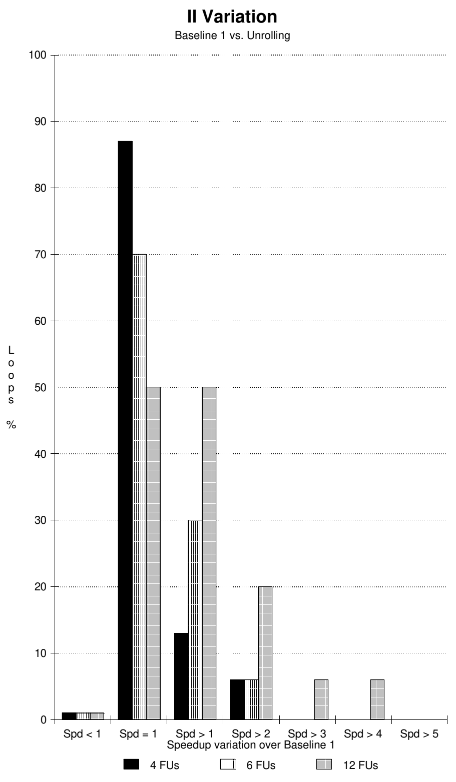


Figure 17: II Variation-Unrolling

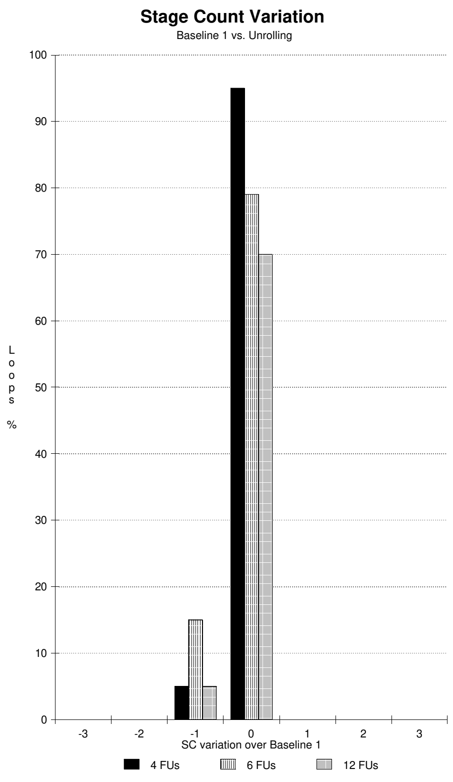


Figure 18: SC Variation-Unrolling

- Conclusions: Loop unrolling produces significant improvements in the use of machine resources, achieving speedups for a considerable fraction of loops. In terms of extra hardware resources it requires a few extra queues and queue positions in some cases, an acceptable overhead considering the benefits generated.

7 Combined Effect of Copy Operations and Loop Unrolling

We have shown that both enhancements discussed in Section 5 and 6 should be incorporated as standard features of the model because they are able to solve a particular problem associated with the QRF (copy operations) and to optimize the parallelism exploitation (loop unrolling). The experimental results obtained when analyzing each of them individually showed that the benefits achieved are more significant than performance penalties or extra machine resources required. However the *combined* effect of both features should be investigated before they are adopted as standard procedures in our framework, which is done is this section. We shall call this new compiling configuration as Baseline 2.

- Number of Queues: The combined effect of copy operations and loop unrolling does not result in a large increase in the number of queues required when compared against the basic configuration, particularly for the most complex loops (Figure 19).
- Name Space - QRF vs. RF: The same effect previously observed is confirmed in this analysis, i.e. the size of the name space using a QRF is less sensitive to the number of FUs, as seen in Figure 20

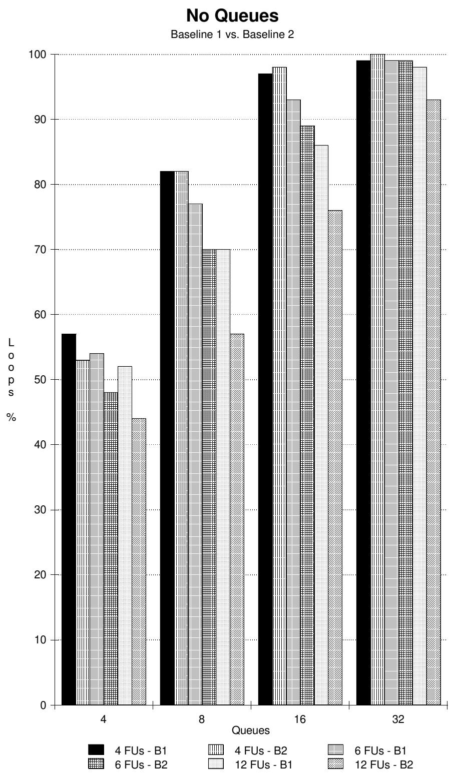


Figure 19: Number of Queues-Baseline 2

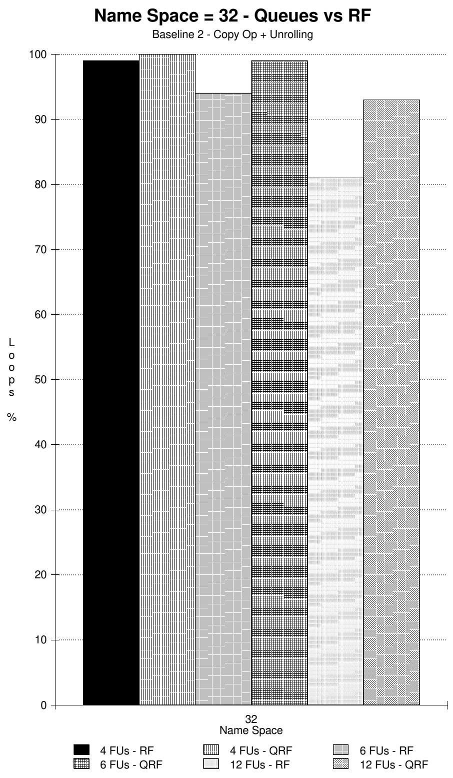


Figure 20: Name Space-Baseline 2

- Machine Requirements Weighed by the Execution Time: We have performed alternative analyses on machine resources using *loop execution times* to weigh the results. Thus instead of assuming every loop of the benchmark as having the same importance we take into account its execution time, which is calculated according to the *iteration counters* associated with a given input data set. As an example, the data in Figure 21 shows, for a 12 FUs machine, the fraction of the total execution time that can be supported by a given number of queues. It can be seen that 32 queues is enough to generate schedules for only 69% of the total execution time, which is considerably worse than the 93% figure obtained when the absolute number of loops is used instead. However we have noticed a few large loops accounting for a significant fraction of the total execution time, so we obtained some figures not considering those loops. If only loops with less than 300 operations are used, leaving all but 3 loops from the original benchmark, the fraction of loops scheduled with 32 queues is 84%. If only loops with less than 100 operations are considered (all but 28 from the original 1258 loops) this number rises to 92%, a figure similar to previous analysis. Two reasons make us believe that the last analysis give us a clearer picture of the machine resources required: Some techniques may be used to restructure the loop body, like *loop distribution* [18], in which case a loop having a large number of operations would be subdivided into smaller ones. Furthermore, just a few loops have in fact more than 100 operations, and even though they take a long time to execute, it is still unclear their importance to the targeted applications.

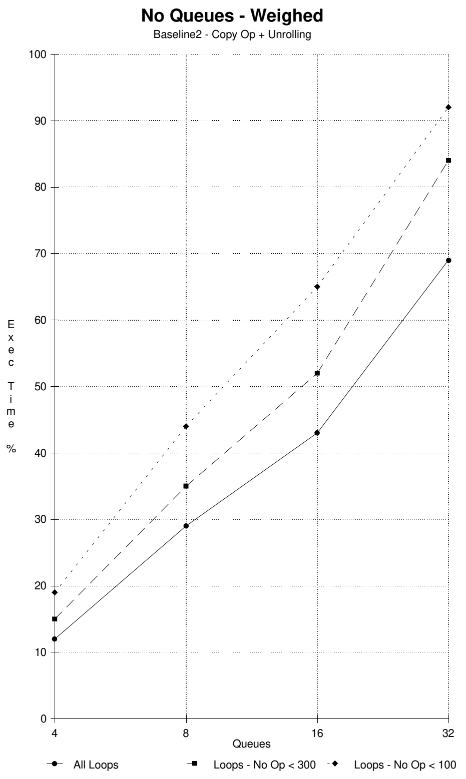
- Total Number of Positions: Only small increases were observed in the total number of queue positions required for the most complex loops, although it is more accentuated for the simple loops, which is mainly due to loop unrolling (Figure 22).

- Initiation Interval Variation: These features do not interfere with each other regarding kernel code execution performance, which means that most of the speedups resulting from loop unrolling are not affected by copy operations. It was also observed that the fraction of loops showing execution delays did not increase by a significant factor, as shows the cumulative distribution of Figure 23.

- Stage Count Variation: This factor exhibits the largest variation in comparison to the basic configuration and also when compared against both features individually, although most of the loop schedules still have the same stage count (Figure 24). When variations occurs they are often a decrease in the stage count value, which may improve the overall performance.

- Scalability - Kernel Speedup: Even though copy operations are used, loop unrolling dramatically improves the benefits resulting from scaling up the machine model, as higher speedup factors are achieved for a larger fraction of loops, as shows the cumulative distribution of Figure 25.

- Scalability - Stage Count Variation: Employing a larger number of FUs changes the stage count for a number of loops, increasing its value in many of these cases (Figure 26). However this increase is much less accentuated than occurred in the basic configuration, as show the average values of SC (SC_{avg}) for each machine model. Further experiments with wider issue machines (15 and 18 FUs) resulted in (SC_{avg}) values similar to the one found for the 12 FUs machine, indicating that this parameter should not compromise scalability (Table 4).



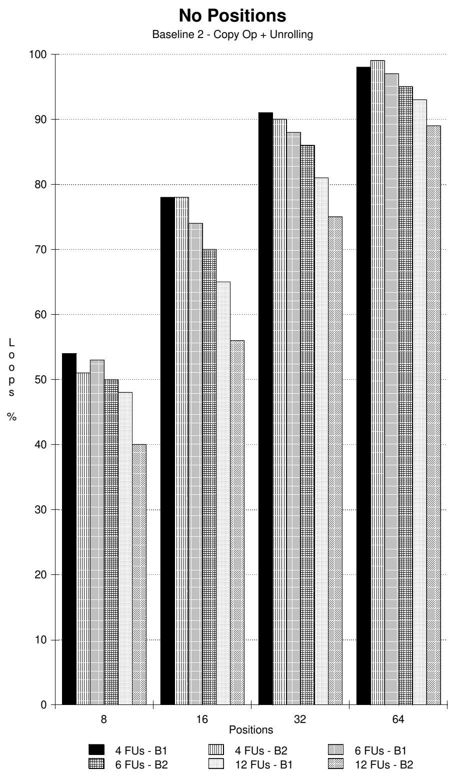


Figure 21: Number of Queues Weighed by Execution Time-Baseline 2

Figure 22: Number of Positions-Baseline 2

No. Functional Units	Average Stage Count
4	3.7
6	4.0
12	4.6
15	4.6
18	4.7

Table 4: Average Stage Count-Baseline 2

- Conclusions: The experimental results have shown that the combined use of copy operations and loop unrolling keep their individual benefits and contributions without significant extra overheads in performance or machine resources. For this reason we shall adopt both of them as standard features of the architecture model being developed and include them in the next experiments, unless otherwise stated.

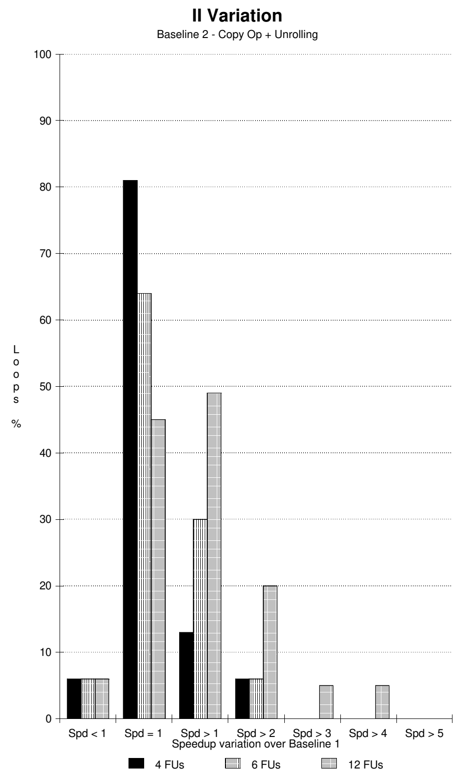


Figure 23: II Variation-Baseline 2

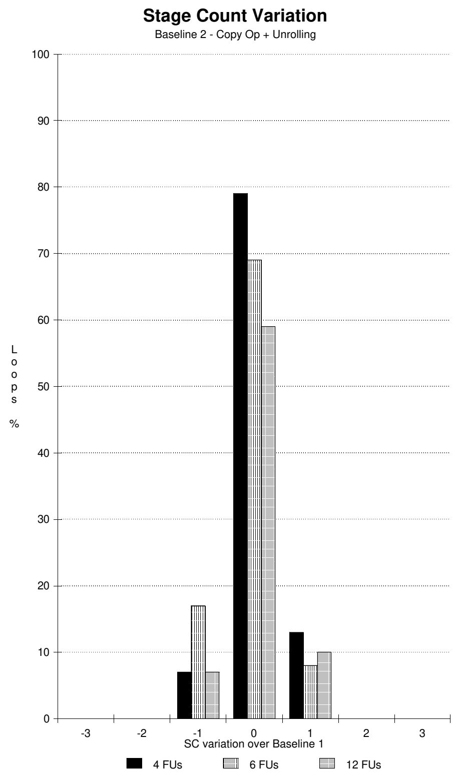


Figure 24: SC Variation-Baseline 2

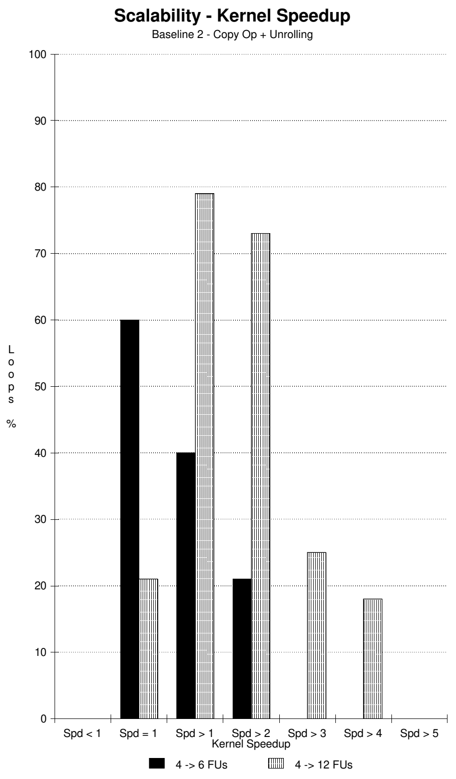


Figure 25: Kernel Speedup-Baseline 2

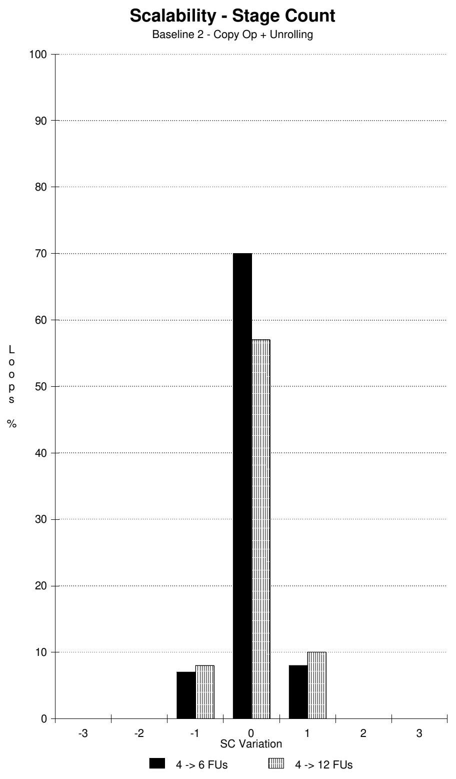


Figure 26: SC Variation-Baseline 2

8 First Experiments with a Clustered Architecture

As already said a single cluster wide-issue VLIW machine would require a highly multiported register file, which unfortunately is difficult to design and implement with the current technology. 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 the achievable MII.

For this first set of experiments we have defined a cluster configuration comprising 3 standard FUs, which are 1 L/S, 1 ADD and 1 MUL, and also an extra FU to support copy operations as shown in Figure 27a. All of them connect to a QRF. We assume that clusters are interconnected by a *bidirectional communication ring*, implemented by means of queue structures (Figure 27b). These communication queues are used to allocate registers as if they were a cluster's 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.

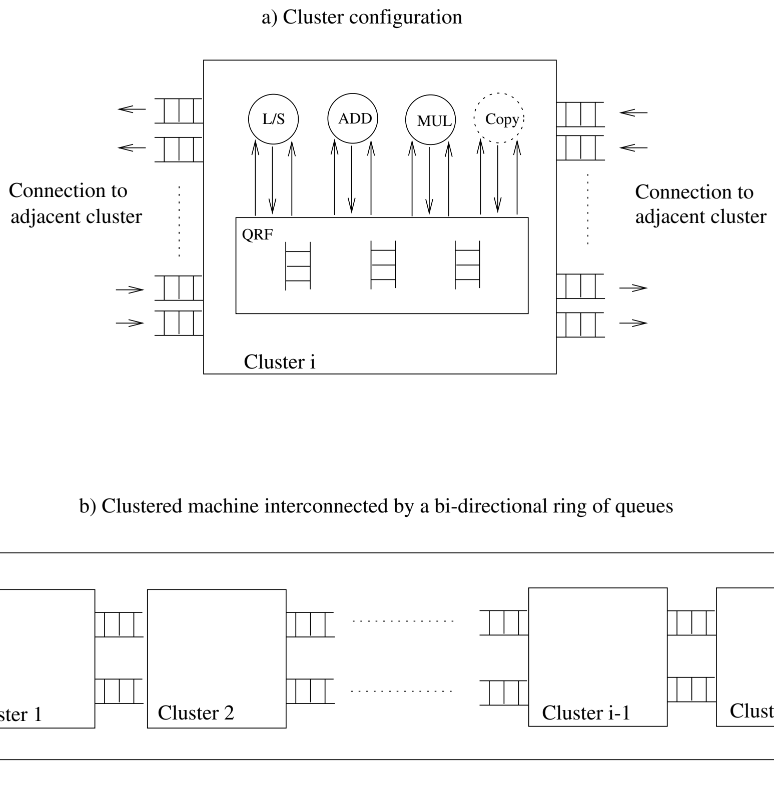

a) Cluster configuration
L/S
ADD
MUL
Copy
Connection to adjacent cluster
Connection to adjacent cluster
QRF
Cluster i
b) Clustered machine interconnected by a bi-directional ring of queues
Cluster 1
Cluster 2
Cluster i-1
Cluster i

Figure 27: Clustered Machine Organization

The partitioning process is carried out by including two simple heuristics to the IMS algorithm:

- An operation is scheduled in an given cluster only if there is no *communication conflicts* with previously scheduled operations, i.e. the clusters to which the communicating operations belong to must be directly interconnected. We are not as yet considering the introduction of operations to transfer a value between indirectly connected clusters, implying that an operation can only send or receive data from an operation scheduled either in the same cluster or in one of the adjacent ones.
- Communication conflicts can prevent an operation from being scheduled in any of the clusters, leading to a *backtracking* process to unschedule conflicting operations. The unscheduled operations are then rescheduled taking into account the new communication constraints arisen from the new partial schedule.

The new heuristics do not increase the complexity of the IMS algorithm as only further constraints are added to the existing ones when choosing a valid time slot to schedule an operation. However the backtracking frequency increases in some cases, requiring a larger II value and consequently reducing 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 the new algorithm version is at distributing operations among clusters for the same II that would be required using a single cluster machine. We have 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, and some results and conclusions are presented as follows:

- Initiation Interval Variation: The data presented in Figure 28 show the fraction of loops scheduled for a clustered machine exhibiting the same II of the schedules for the corresponding single cluster machine. There is no increase in the II value for 95% of 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 show that the IMS partitioning scheme is able to schedule for a clustered machine requiring either none or just a few extra II cycles in most of the cases. It should be noticed that this performance degradation occurs when comparing to an ideal single cluster machine whose realizability is problematic in terms of the register file implementation, which should not be the case if a queue register file is used instead. However the results indicate that the partitioning scheme adopted tends to produce poorer results as the number of clusters increases, leading us to seek alternative methods.
- Stage Count Variation: As seen in Figure 29 the stage count remains the same for a large fraction of the loops. Most of the variations observed are reductions in its value, suggesting that the performance of a clustered machine should not be significantly influenced by variations in the stage count.

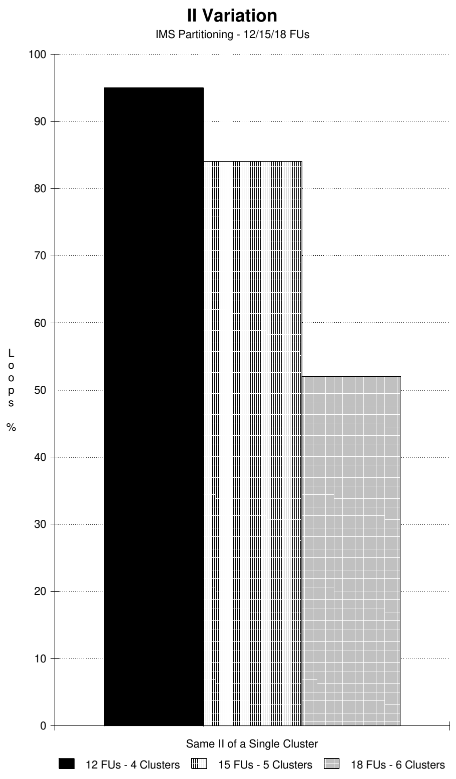


Figure 28: II Variation-IMS Partitioning

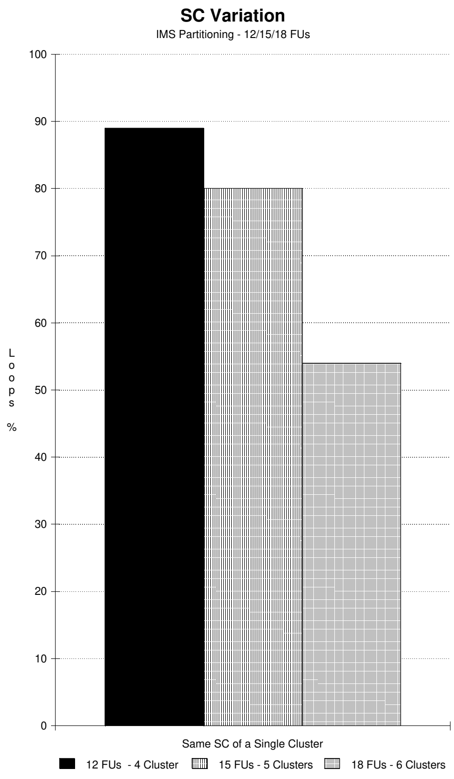


Figure 29: SC Variation-IMS Partitioning

- Scalability - Kernel Speedup: When a clustered machine model goes from 4 to 5 clusters (12 to 15 FUs) performance gains in terms of kernel speedup are observed in 56% of loops. If an extra cluster is used, which increases the number of FUs to 18, the fraction of loops that benefit from this machine upgrading is 24%, as seen in Figure 30. The worse performance gains observed in the second case is mainly due to the inability of the partitioning process to schedule for a larger number of clusters.
- Scalability - Stage Count: The values of stage count remains the same for most of the loops as the clustered machine model scales up, as show the average values on Table 5, suggesting that is not an issue for the partitioning strategy.
- Number of Queues: The diagram of Figure 31 shows the queue requirements of a 4 cluster machine, which are individually presented for *every cluster* and for *every communication link* between clusters. Those figures suggest the required cluster configuration would comprise 8 queues for the private QRF and another 16 queues to implement the communication ring (8 to be used in each direction), as shown in Figure 32. A small fraction of loops would require additional resources, however we believe there are still some possibilities to improve the partitioning algorithm, which may leave just a few loops demanding additional techniques like the introduction

No. Clusters	Average Stage Count
4	4.6
5	4.1
6	4.5

Table 5: Average Stage Count-IMS Partitioning

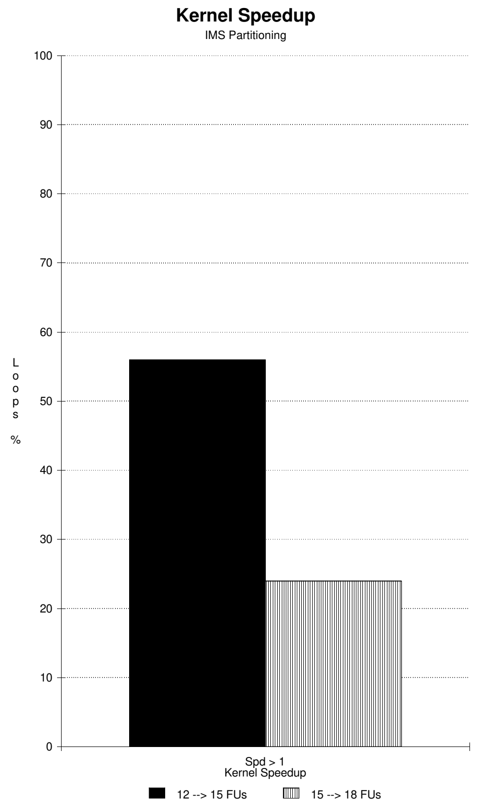


Figure 30: IMS Partitioning-Kernel Speedup

of spill code. An improvement that should be easy to implement is the inclusion of further heuristics to balance the communication among clusters. An interesting observation is that this cluster configuration should also suffice for any of the three machine models analysed (12, 15, and 18 FUs), indicating once again that this approach favours scalability.

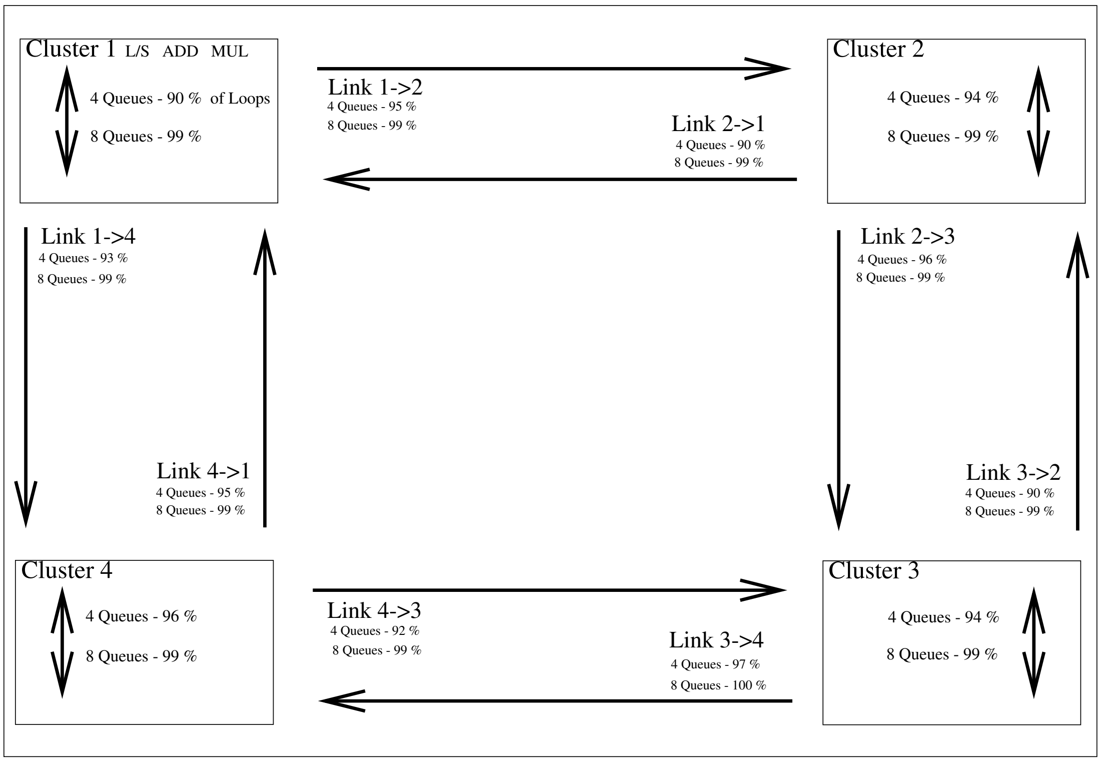


Figure 31: Example: Queue Requirements-Four-Cluster Machine

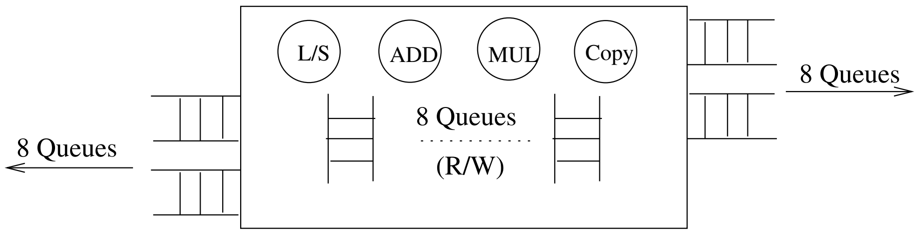


Figure 32: Suggested Cluster Configuration

- **Parallelism Exploitation**: We have performed static and dynamic analysis on the *average number of operations issued per cycle.* The static issue, $Issue_{static}$, accounts for the number of instructions issued at the kernel phase, being calculated through the following expression:

$$Issue_{static} = \frac{\sum (No\,Ops - Copy\,Ops)}{\sum II} \tag{8}$$

The dynamic issue, $Issue_{dynamic}$, take into account the number of iterations performed on each loop body to estimate its execution time, including the prologue and epilogue phases, as show the expression:

$$Issue_{dynamic} = \frac{\sum (No\,Ops - Copy\,Ops) \times Iteration\,Counter}{\sum Execution\,Time} \tag{9}$$

One of the analysis considered all loops of the benchmark, as seen in Figure 33. Those results are compromised by the fact that many of the loops are simply not able to take advantage of the extra machine resources available as they are constrained by recurrent circuits in the loop body. A significant improvement in those figures would require compilation techniques able to identify further instruction level parallelism. 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. In this case only those loops having $ResMII \geq RecMII$ were taken into account. The data on Table 6 shows the fraction of loops that were found to be resource constrained for each of the machine models considered. It should be noticed that every time the number of FUs increases a new set of loops become recurrence constrained, thus preventing them from benefiting of further machine upgrades. As expected the results are significantly better than in the first case, as show the data in Figure 34. One way to further improve those figures would be the design of non-dedicated functional units, able to execute either an addition or multiplication, for instance. 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 for a more sophisticated scheme. 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 explicited by dynamic analysis. This is more noticeable for clustered machines because these large loops can be scheduled without performance degradation due to the partitioning algorithm, which further emphasizes their influence on the overall results.

No. Functional Units	Fraction of Loops
4	85%
6	78%
12	72%
15	70%
18	69%

Table 6: Resource Constrained Loops-Baseline 2

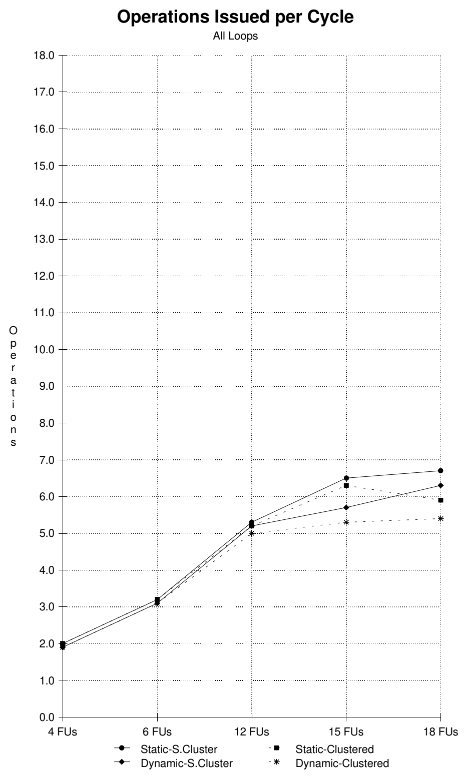


Figure 33: Operations/Cycle-All Loops

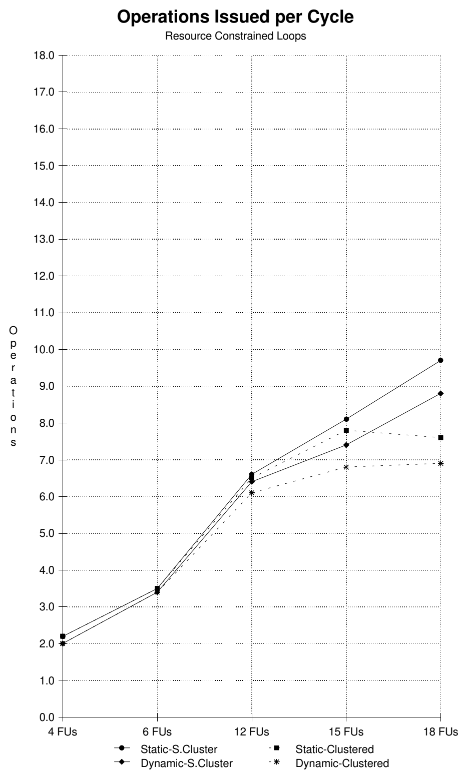


Figure 34: Operations/Cycle-Resource Constrained Loops

- Conclusions: The simple partitioning scheme presented has produced satisfactory results for the 4 and 5 clusters machine model. However it has to be improved to partition for 6 or more clusters. Another option would be the development of a new approach, employing a more sophisticated partitioning algorithm and a *cost function* to minimize the number of extra II cycles and the inter-cluster communication. The basic reason to have an increase in the II is the introduction of a large number of *move operations* to allow the communication between non-directly connected clusters, which has not been done as yet. A new partitioning scheme should minimize the frequency of this type of communication, trying to use only a number of move operations compatible with the MII previously determined. In terms of machine resources these experiments have been useful to estimate the number of queues required to implement a cluster and its communication structure, providing some quantitative data that should not change dramatically if another partitioning algorithm is used.

9 Some Alternatives to Implement a Queue Register File

So far we have referred to a queue register file without any information regarding its design or implementation. This section presents a preliminary discussion on the issue, which will be further extended in the near future in order to propose a more realistic architecture model. The limitations imposed by silicon area, access time and power consumption have recently motivated the development of alternative organizations to a conventional multi-ported register file. Aloqeely proposes *sequencers*, such as queues or stacks, to organize memory elements [2], being the register allocator responsibility to ensure that data values are read in written in the proper order. The basic approach he uses when working with queues is to allocate values having the same lifetime to the same queue, as long as the cycles when read operations occurs are not coincident, a characteristic often found in applications with a high degree of regularity like digital signal processing. The sequencers are implemented using shifting registers, which imposes a latency constraint between write and read operations of the same value. This alternative also presents a high power consumption for large memory sizes, a problem that may be minimized if arrays of memory cells and a sequential register (used as a 'pointer') are used instead, as proposed by Heubi [15]. A version of this structure, called *sequential read-write memory (SRWM)* has been investigated by Gerez [14], who has concluded that SRWM has some advantages over conventional memory structures regarding address decoding and controller implementation.

We now present three alternatives for the design of a QRF suitable to be used by the VLIW architecture being developed. We have discarded the possibility of using shift registers to implement a queue as this would impose an unacceptable constraint on the register allocator due to latency periods between read and write operations to a queue.

9.1 Allocation of a Queue to a Dual Register File

Assuming that a *dual register file (DRF)* is capable of performing one read and one write operation simultaneously, it could be used to allocate *one* of the 'logical' queues used by the schedule, as shows the scheme of Figure 35. This is an attractive alternative in the sense that the hardware structure exactly matches the machine model targeted by the compiler, however it presents a number of complicating factors that must be addressed to adopt it:

- Every FU must be connected to every queue, which would require a possibly complex crossbar connection between FUs and DRFs.
- The number of queues would be fixed, being defined by the number of dual register files available.
- The number of queue positions would also be fixed, defined by the size of the dual register file.

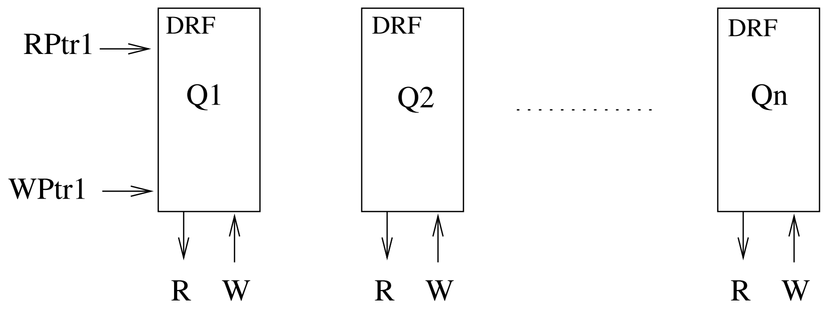


Figure 35: One Queue Allocated to a DRF

9.2 Allocation of Groups of Queues to a Dual Register File

Some of the problems presented by the first alternative (Section 9.1) may be minimized if more than one queue could be allocated to the same dual register file (Figure 36). This approach still requires a crossbar connection between FUs and DRFs, however a smaller number of them would be used, simplifying the connection. It would also allow some flexibility in the number and size of each queue, which is an interesting feature to be exploited by the register allocator. However the register allocation procedure would require an extra step to allocate queues to DRFs in order to prevent access conflicts, a procedure that may result in further complications when scheduling complex loops.

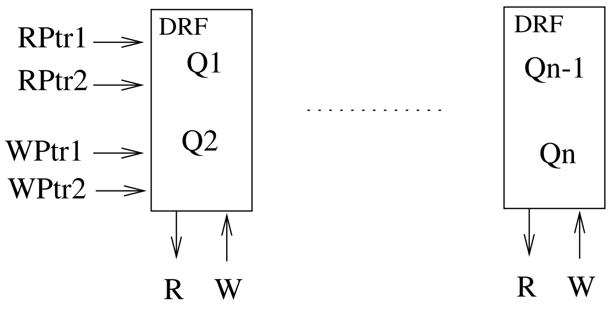


Figure 36: Groups of Queues Allocated to a DRF

9.3 Allocation of All Queues to a Multiport Register File

Another alternative would be the allocation of every queue to a multiport register file, as seen in Figure 37. This approach presents the maximum flexibility in terms of number of queues and queue positions, which would be limited only by the size of the RF and by the pointer structure that control the access to queues. In terms of connections between FUs and queues it might be the best solution as the number of access ports depends only on the number of functional units (typically three ports per FU). If a clustered machine model is adopted the total number of access ports required by each QRF should not be large as extra FUs would be possibly included in a new cluster, and not in an existing one. The issue that remains to be solved regards the complexity involved in the implementation of such scheme. In a first attempt to understand this problem we have derived expressions showing that the *area* of this QRF design grows proportionally to the *square* of the number of FUs, and the access delay is proportional to the number of FUs, as showed by the diagrams and expressions of Figure 38. Although the analysis has been done only considering write access ports, the inclusion of read ports should not change these conclusions. We should further extend the design and analysis of this approach before presenting it as feasible solution, however it seems to be an interesting alternative particularly if the number of FUs sharing a QRF is small, which should be the case of a clustered machine.

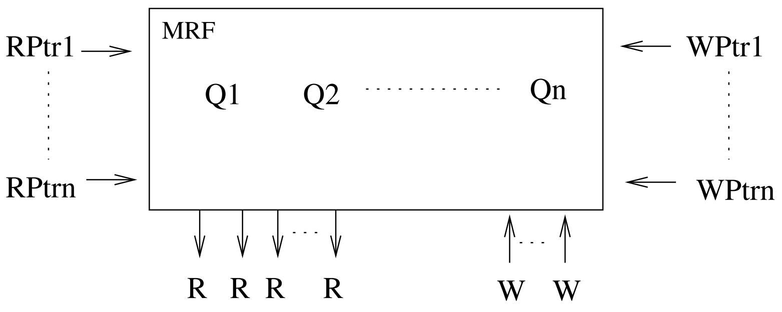


Figure 37: All Queues Allocated to a Multiport RF

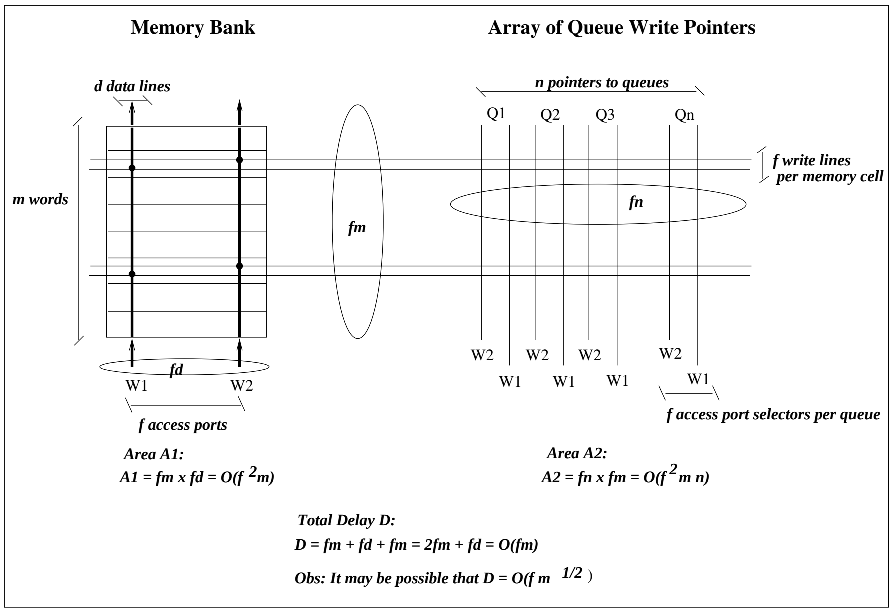

Memory Bank
Array of Queue Write Pointers
d data lines
m words
fd
W1
W2
f access ports
fm
n pointers to queues
Q1
Q2
Q3
Qn
f write lines per memory cell
fn
W2
W1
f access port selectors per queue
Area A1:
A1 = fm x fd = O(f 2m)
Area A2:
A2 = fn x fm = O(f 2m n)
Total Delay D:
D = fm + fd + fm = 2fm + fd = O(fm)
Obs: It may be possible that D = O(f m 1/2)

Figure 38: Complexity Analysis of a QRF Implemented on a Multiport RF

10 Next Steps

A number of improvements should be done in order to produce a more realistic machine design and corresponding experimental analysis. We understand that a new partitioning procedure and a scheme to deal with loop invariants are the most important features at this stage, so we have already started to investigate these topics, although we also plan to deal with other topics described bellow in the near future.

- New Partitioning Procedure: We have started the development of a more robust partitioning algorithm able to produce high quality schedules for machine organizations comprising a larger number of clusters. The basic approach to the problem can be summarized by the following steps:

  1. Calculate MII considering a single cluster machine, which should be the lower bound on the MII for the clustered machine.
  2. Partition the ddg in such a way that minimizes inter-cluster communication costs and also makes, for every cluster i, $MII_i \cong MII$. An efficient partitioning algorithm should be used, incorporating a *cost function* to minimize the number of extra II cycles and the inter-cluster communication. The basic reason to have an increase in the II is the introduction of a large number of *move operations* to allow the communication between non-directly connected clusters. Thus the partitioning scheme should minimize the frequency of this type of communication, trying to use only a number of move operations compatible with the MII previously determined.
  3. Schedule operations according to the resulting partitioning.
  4. If necessary move operations after scheduling in order to overcame eventual machine constraints.

- Allocation of loop invariants: The QRF adopted implies that a value can be read only once, which is a hard constraint to deal with loop invariants as they are used several times during the loop execution. We have discussed two preliminary alternatives to deal with the problem, which should be further investigated by means of experimental analysis before adopting any of them:

  - Treat loop invariants as a loop variant that should be write back to a queue after its use, using copy operations to implement the process.
  - Implement a *private register file (PRF)* to each FU (Figure 39), which would be used mainly to store loop invariants, or alternatively loop variants in the absence of loop invariants.

- New machine model: It would be interesting to consider a more realistic machine model, particularly regarding FUs latency, pipeline stages, and register file capacity, using as a reference one of the latest commercial available such as the DEC Alpha-21264.

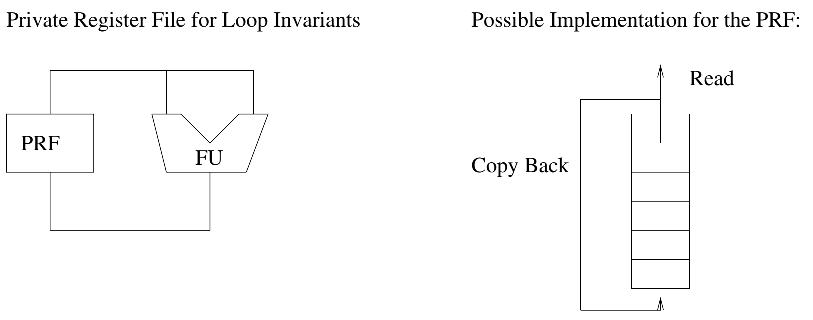

Private Register File for Loop Invariants
PRF
FU
Possible Implementation for the PRF:
Read
Copy Back

Figure 39: Hardware to Support Allocation of Loop Invariants

- Finite resources: A further stage of this work should assume a fixed number of resources such as number and capacity of queues. Possible problems arisen could be dealt with techniques like graph coloring and the introduction of spill code, for instance.

11 Conclusion

This report has shown some improvements added to a 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. We should further investigate the best hardware organization to support this scheme. The use of loop unrolling resulted in dramatic improvements in loop execution and parallelism exploitation, which was accomplished without a significant increase in the number of machine resources required. We understand it as being a fundamental feature to optimize the use of wide-issue machines. First 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 simple 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. Finally we have started to work on the design of possible hardware organizations to implement a queue register file. In order to further improve this model we are currently working on new partitioning techniques, strategies to deal with loop invariants, and also refining some of the hardware specifications adopted.

References

[1] 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.

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

[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] 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, November 1988.

[6] 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.

[7] G. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings ACM SIGPLAN Symp. on Compiler Construction*, 1982.

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

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

[18] D. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, March 1977.

[19] 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.

[20] 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.

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

[22] 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.

[23] K. Mehlhorn and St. Naher. *The LEDA Platform of Combinatorial and Geometric Computing.* to apperar with Cambridge University Press, 1997.

[24] B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the MICRO-27 - The 27th Annual International Symposium on Microarchitecture*, November 1994.

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

[26] 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.

[27] 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.

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

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

[30] F. Sanchez. *Loop Pipelining with Resource and Timing Constraints.* PhD thesis, UPC - Universitat Politecnica de Catalunya, October 1995.