Non-Consistent Dual Register Files to Reduce Register Pressure *

Josep Llosa, Mateo Valero and Eduard Ayguade
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, SPAIN

Abstract

The continuous grow on instruction level parallelism offered by microprocessors requires a large register file and a large number of ports to access it. This paper presents the *non-consistent dual register file*, an alternative implementation and management of the register file. *Non-consistent dual register files* support the bandwidth demands and the high register requirements, without penalizing neither access time nor implementation cost. The proposal is evaluated for software pipelined loops and compared against a unified register file. Empirical results show improvements on performance and a noticeable reduction of the density of memory traffic due to a reduction of the spill code. The spill code can in general increase the minimum initiation interval and decrease loop performance. Additional improvements can be obtained when the operations are scheduled having in mind the register file organization proposed in this paper.

Keywords: VLIW and superscalar processors, software pipelining, register file organization, register allocation, spill code.

1 Introduction

Current high-performance floating-point microprocessors try to maximize the exploitable parallelism by either heavily pipelining functional units[1][2] or by making aggressive use of parallelism[3][4]. It is expected that future high-performance microprocessors will make extensive use of both techniques. To effectively exploit this amount of available parallelism new processor organizations and scheduling techniques are required.

Software pipelining[5] is a loop scheduling technique that extracts parallelism from loops by overlapping the execution of several consecutive iterations. Finding the optimal solution is an NP-complete problem and there exist several works that propose and evaluate different heuristic strategies to perform software pipelining[6][7].

The drawback of aggressive scheduling techniques such as software pipelining is that they increase register requirements compared to less aggressive and less effective scheduling techniques. In addition, increasing either the stages of functional units or the number of functional units, which are the current trends in microprocessor design, tends to increase the number of registers required by software pipelined loops[8][9]. When the number of registers required in a loop is larger than the available number of registers, spill code has to be introduced to reduce register usage. This spill code increases memory traffic and can reduce performance.

Usually, registers are organized in a multiported register file as shown in Figure 1a. Each port of each functional unit has access to all the registers of the multiported register file. This register file organization can be expensive and increase processor cycle time when a large number of registers and ports are required. In order to reduce the complexity of the register file some microprocessors, such as the Power 2 [3], implement the register file with two register subfiles with the same number of registers, same number of write ports, but half the number of read ports into each register subfile (see Figure 1b). This implementation, which we name *consistent dual register file*, is totally transparent to the user/compiler because both register subfiles are consistent, i.e both store exactly the same value in the same registers.

In this paper we modify the *consistent dual register file* organization so that each subfile can be accessed independently of the other and store different values; this organization is shown in Figure 1c and it is named *non-consistent dual register file* organization. Due to computational requirements, some values will be copied into both register subfiles as in the *consistent dual register file* organization; other values will be stored in just one of the register subfiles. In order to reduce the number of values stored in both register subfiles, and to balance the number of registers required in each subfile, we also evaluate the effectiveness of swapping operations. To evaluate the register file organization we have used a set of loops from the Perfect Club Benchmark suite [10].

The outline of the paper is as follows. Section 2 introduces the architecture we are assuming and makes a brief overview of software pipelining, register allocation and terminology associated with them. Section 3 presents the observations that motivated our proposal. In Section 4 the *non-consistent register file* organization is presented; an example loop is scheduled and it is used to show how our organization can reduce register requirements. Section 5 presents the experiments performed in order to evaluate the proposal and

*This work was supported by the Ministry of Education of Spain under the contract TIC 880/92, by ESPRIT 6634 Basic Research Action (APPARC) and by the CEPBA (European Center for Parallelism of Barcelona).

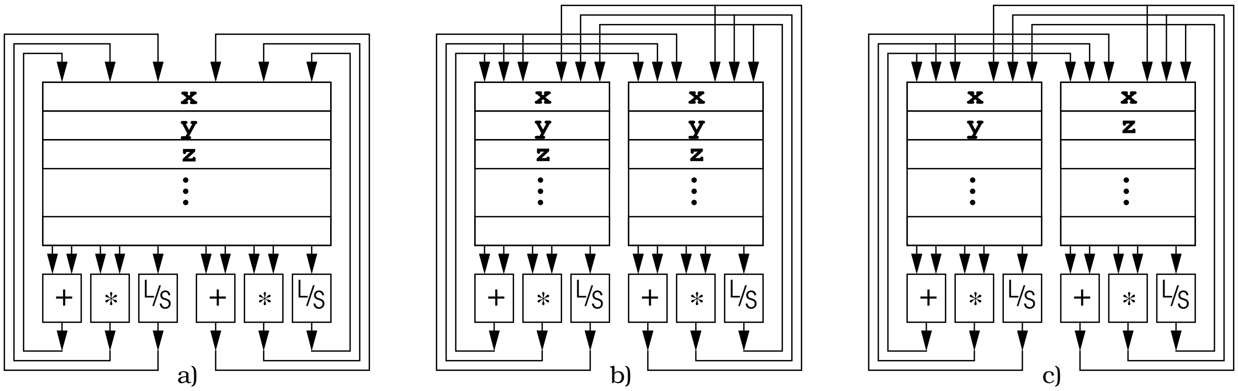

Figure 1: a) Multiported register file. b) Consistent dual register file. c) Non-consistent dual register file.

presents some relevant results. Section 6 summarizes and presents some conclusions.

2 Target Architecture, Scheduling and Register Allocation

The target machine model we are dealing with is a VLIW floating-point processor, similar in some aspects to Cydrome's Cydra 5 [11]. In order to abstract the effects of the memory subsystem, we have assumed that the VLIW processor is the floating point unit of a conventional microprocessor or of a decoupled architecture[12].

Architectural support for software pipelined loops without code replication (such as rotating register files and predicated execution) is assumed[13]. The floating register file is partitioned into two sets of registers: the rotating register file contains rotating floating-point registers to store loop variants, and the general register file contains loop invariant floats. The addresses and integer values are assumed to be stored in the address processor of a decoupled architecture or in the case of a conventional VLIW, in the integer unit.

Modulo scheduling[7] has been used to schedule the dependence graphs of the loops. It is a technique that overlaps iterations of the innermost loop. The number of cycles between the initiation of successive iterations in a modulo-scheduled loop is termed the *initiation interval (II)*.

Once a loop has been scheduled, register allocation determines its register requirements. Historically, register allocation has been performed adopting Chaitin's technique based on graph coloring[14]. Register allocation for software pipelined loops presents additional problems leading to unconventional solutions. How to allocate registers for modulo-scheduled loops is beyond the scope of this paper (for an extensive discussion of the problem see [15]). The Wands Only strategy combined with the First Fit allocation schema have been chosen to allocate registers. Wands Only is the strategy that has the lowest empirical complexity, and the one that obtains the more optimal results in terms of number of registers. For this strategy all the allocation schemes have similar results, but First Fit has been selected due to its simplicity.

The register allocator assumed that lifetime of a value starts when the producer operation is issued, and ends when all the consumer operations finish. This definition of lifetime is required if the code has to be interruptible and re-startable with a hardware model in which operations that have been issued always go to completion before the interrupt is handled.

Lifetimes corresponds either to loop-invariant variables or to loop-variant variables. Loop invariants are assumed to have already been allocated in the (non rotating) general register file. This paper concentrates only on floating-point loop variants, because the number of registers to store loop invariants remain constant independently of the architecture. The register requirements for loop variants increase with the latency (stages) of functional units and with the number of functional units. As shown in [16] loop invariants require no more than 16 registers for 98% of loops[1] and only 3 loops out of 1525 use more than 32 registers for loop invariants. So, hereon "registers" refers to the rotating register file, where loop variants are stored.

3 Motivation of the Proposal

3.1 Register Requirements of Pipelined Loops

The register requirements of loop variants for floating-point intensive pipelined loops and their effect on performance as a function of the stages and number of functional units have been studied in [9]. In this paper, the authors consider a variety of architectures ranging from one adder and one multiplier of latency 3 to two adders and two multipliers of latency 6, one store port and two load ports. Table 1 shows some of the results obtained. Notice that 0.3% of the loops[2] require more than 64 registers for loop variants (in order to avoid spill code) in configurations with low inherent parallelism (P1L3). However, when the architecture becomes more aggressive in terms of

[1] Belonging to Lawrence Livermore Loops, The SPEC89 Fortran benchmarks, and The Perfect Club codes.

[2] The paper uses 795 loops belonging to the Perfect Club Benchmark Suite.

	% of Loops			% of Cycles		
	16	32	64	16	32	64
P1L3	63.1%	95.2%	99.7%	14.5%	66.4%	99.8%
P2L3	60.4%	84.9%	99.2%	22.9%	41.1%	74.6%
P1L6	43.8%	72.2%	97.6%	8.6%	21.1%	68.4%
P2L6	43.0%	65.4%	89.4%	13.9%	24.4%	50.9%

Table 1: Percentage of loops that can be allocated without spilling with 16, 32 and 64 registers and percentage of cycles those loops represent. PxLy denotes a configuration with x adders of latency y, x multipliers of latency y, one store port and two load ports..

parallelism (P2L6) 10.6% of the loops, representing 49.1% of the dynamic execution time, require more than 64 registers. For the most aggressive configuration (P2L6), limiting the number of registers to 64 registers leads to a reduction in performance (for the whole collection of loops) of 6% due to the addition of spill code. And, when the number of registers is limited to 32, the performance experimented a reduction of 31%.

3.2 Register Files

Regardless of the number of registers required, an architecture with a large number of functional units will require a register file with a large number of ports. The number of registers and the number of ports have an important effect on the area and the access time of the register file.

The area of a multiported register file can be modeled as a linear function of the number of registers, the number of bits per register, and as a quadratic function of the number of ports [17]. The access time can be modeled as a logarithmic function of the number of read ports and as a logarithmic function of the number of registers [18].

VLSI technology continues improving at a phenomenal rate. Since the introduction of the first microprocessor, the number of transistors per processor chip has doubled every two years, so the area required by the register file is not a big problem. On the other hand the number of read ports and the number of registers limit the access time to the register file, which can limit the cycle time of the processor. The first step that was proposed in this direction was to split the register file into an integer file and a floating-point file, each file having fewer ports. This partitioning can reduce the bandwidth requirements of each register file, but microprocessors are increasing the number of functional units, adding more port requirements to the register files. To overcome this problem, some current microprocessors use an implementation trick that consists on duplicating the register file. For instance, the IBM Power 2 implements a register file of 8 read ports and 4 write ports using two register subfiles (with 4 read ports and 4 write ports each), one for each execution unit[3]. This implementation reduces access time since each execution unit has to access a 4 read port register file. This dual register file has two identical copies of each register (as shown in Figure 1b). When a functional unit produces a result, it is written in the same register of both register subfiles.

[3] An execution unit is a cluster of several functional units. For instance the integer execution unit of the Power 2 contains an adder, a logic functional unit, a multiply/divide unit and a load/store unit.

3.3 Register Instances

Each time a datum is written into a register, a new *register instance* is created. Succeeding reads to the register *use* the latest register instance. Normally, the creation of each register instance requires a write access to the register file, and each use of a register instance requires a read access to the register file.

A large number of register instances are used only once [19]. This single use property is not surprising for two reasons. First, many register instances, specially in floating-point intensive loops, are created to hold intermediate results that do not appear in the source program, and are used only once or a few times. Second, even if a particular frequently-accessed variable may be accessed repeatedly with reads and writes, each new value assigned to the variable is not.

The single use property has been used in vector machines in the form of chaining to decrease the overall latency and increase the throughput of a sequence of vector operations [20]. More recently superscalar architectures such as the IBM RS/6000 [21] and the MIPS R8000 [4] attempt to exploit this phenomenon by replacing a sequence of operations that produce single use results by higher strength operations (e.g., multiply-add fuse units).

4 Non-Consistent Dual Register Files

In this section we propose an effective utilization of dual register files based on the previously exposed facts: (1) Software pipelined loops have high register requirements. (2) Big multiported register files are expensive in area and can penalize cycle time. (3) Most of the register instances are read only once.

There are other related works that propose different register file organizations. For example partitioned register files with limited connectivity [18] where each cluster of functional units has a private register file, and communication among clusters is accomplished through a number of inter-cluster data transfer buses. In [22], an asymmetric organization consisting of a small high bandwidth multiported register subfile and one or more low bandwidth port-limited register subfiles called "sacks" is presented.

In the *non-consistent dual register file* organization we have a duplicated (consistent) copy of some register instances that are going to be consumed by the two clusters (a cluster is a set of functional units that use the same register subfile). For the rest of register instances that are going to be used by only one of the two clusters, we will use only registers in the register subfile of that cluster (see Figure 1c). So, in our model each register subfile has simultaneously global and private registers.

In this paper *non-consistent register files* are evaluated for software pipelined loops in VLIW architectures because software pipelining imposes higher register requirements than other scheduling techniques and loops are the computational intensive parts of scientific computations. However, this technique could be applied to other scheduling techniques and to other parts of the code. In addition to VLIW other processor implementations (such as superscalar microprocessors) would benefit from having *non-consistent register files*.

4.1 Example

In this section we show with a simple loop how register allocation can be performed when a non-consistent dual register file is available. We also show how the scheduling of operations on the appropriate cluster can improve register usage. Figure 2a shows the source code and Figure 2b shows the corresponding data dependence graph for the loop.

a)
```
DO I=1,N
     z(i) = x(i)+t*(y(i)+r*x(i))
ENDDO
```

b)

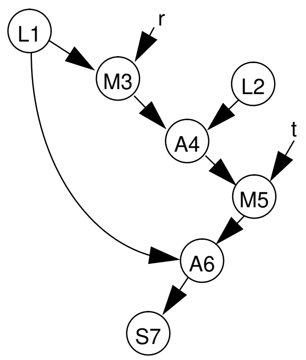

Figure 2: a) Fortran loop, b) data dependence graph of the loop.

Scheduling. Figure 3 shows a scheduling for this loop when Modulo Scheduling [7] is used assuming a hypothetical machine with two adders, two multipliers and four load/store units. The adders and the multipliers are fully pipelined and have 3 cycles of latency; the load/store units have one cycle of latency. Note that the functional units have been organized into two clusters of one adder, one multiplier and two load/store units each one; this partitioning will help when we will consider the case of a dual register file.

VALUE	L1	L2	M3	A4	M5	A6
Start	1	1	2	5	8	11
End	13	7	7	10	13	14
Lifetime	13	7	6	6	6	4

Table 2: Lifetimes of all loop variants for the example loop

The minimum number of additions and multiplications required to perform one iteration of the loop is two (A4, A6, M3 and M5 in Figure 2.b). This means that the saturated resources are the adder and the multiplier. The minimum instruction 'packed' size to execute this loop is therefore one. The schedule is partitioned into 14 pipestages, each of which is 1 cycle in duration, i.e. the II of the scheduling is 1. Fourteen concurrent iterations of the loop are required to saturate the multiplier and the adder. Note that although a single iteration takes 14 cycles to execute, a new iteration can be started each cycle.

In a modulo scheduled loop, the same pattern of operations is executed in each stage of the steady state portion of the pipelined execution. This behavior can be achieved by looping on a piece of code that is termed the kernel. Figure 4 shows the code for the kernel. Also, shown in figure 4 is the stage of the schedule from which each operation comes from. Operations in the kernel code which are from distinct stages are from distinct iterations of the original loop.

Lifetimes. The code in Figure 4 is not semantically correct if no additional support is provided to ensure that successive outputs of an operation can be kept in distinct registers. Consider the operation L1 which at cycle 1 loads a new value of vector x into a register. The lifetime of this value extends to cycle 13 when it is used for the last time [4]. However, at cycle 2 the same operation is executed again and will overwrite the previous value of the register while it is still live, yielding to an incorrect result. One approach to fix this problem is to provide some form of register renaming (rotating register files [13]) so that successive definitions of the same "virtual" register actually use distinct physical registers.

In the resulting scheduling the value produced by operation L1 has a lifetime of 13 cycles and a new value is produced every II=1 cycles, so at least 13 physical registers are required to store the successive values produced by operation L1. Table 2 summarizes the start time and the end time of all values produced by the operations of the loop, and the corresponding lifetime (which in the special case of II=1 corresponds to the physical registers required by each value). We will not consider the loop invariants r and t.

Register allocation with a unified register file. If a rotating register file is used to store loop variants,

[4] Actually it is used for the last time in cycle 11, but we defined the lifetime of a value from the start of the producer operation to the end of the last consumer operation

Pipestage	LEFT CLUSTER ADD	MUL	L/S	L/S	RIGHT CLUSTER ADD	MUL	L/S	L/S
1			L1	L2				
2		M3						
3								
4								
5	A4							
6								
7								
8						M5		
9								
10								
11					A6			
12								
13								
14							S7	

Figure 3: Modulo schedule for the example loop.

LEFT CLUSTER				RIGHT CLUSTER			
[5] A4	[2] M3	[1] L1	[1] L2	[11] A6	[8] M5	[14] S7	nop

Figure 4: Kernel code after modulo scheduling (numbers in brackets represent the stage each operation comes from).

a new definition of each of these loop variants will be created every iteration. This means that for each value there are several alive concurrent values corresponding to successive iterations. In this particular case (where II=1) there will be as many concurrent alive values as the lifetime of the variable. So, the total register requirements of this loop schedule are the sum of lifetimes of all the values [5]. In this case the schedule requires at least 42 registers if a conventional register file is used.

Register allocation with a non-consistent dual register file. Let's now consider the case of a non-consistent register file. For this purpose assume that we have two clusters with one adder, one multiplier and two load/store units each one. In Figures 3 and 4 we have divided the operations into two groups that correspond to the "left" cluster and to the "right" cluster.

Now we have to decide which values have to be allocated as global values (using the same registers in both clusters) and which values can be allocated as local values (using registers only on one of the clusters). For instance consider operation L1. The result produced by this operation is used by operations M3 and A6. Operation M3 has been scheduled in the "left" cluster and operation A6 has been scheduled in the "right" cluster, so the values produced by L1 have to be allocated as global values. On the other hand consider operations M3 and A4. The results of M3 are used by operation A4; since A4 has been scheduled in the "left" cluster, the values produced by M3 could be allocated as left-only values. The results of A4 are used by operation M5; since M5 has been scheduled in the "right" cluster, the values produced by A4 can be allocated as right-only values.

[5]We have chosen this example because it is very simple to calculate the registers required by the schedule. For an extensive discussion of the register allocation problem for software pipelined loops see [15].

VALUE	L1	L2	M3	A4	M5	A6
Allocation	GL	LO	LO	RO	RO	RO
Lifetime	13	7	6	6	6	4

Table 3: Allocation requirements of values for example loop.

Table 3 shows, for all the operations of the schedule, if the corresponding value has to be allocated as global (GL), left-only (LO) or right-only (RO). Based on this table, the loop requires 13 global registers, 13 left-only registers and 16 right-only registers. So, the total register requirements correspond to the maximum of both clusters. In this example the "right" cluster has to be able to allocate 29 registers (13 global + 16 local).

Swapping of operations. Finally we show that the registers required can be reduced even more if the operations are properly scheduled to clusters. Two operations in different clusters can be swapped if they use the same kind of functional unit. Swapping two operations has two objectives, both of them can help in reducing the registers required by the schedule.

- Balance the number of left-only registers with the number of right-only registers.
- Reduce the number of global registers.

In our example loop, value produced by operation L1 is consumed by two operations (M3 and A6) that

LEFT CLUSTER				RIGHT CLUSTER			
[11] A6	[2] M3	[1] L1	[1] L2	[5] A4	[8] M5	[14] S7	nop

Figure 5: Kernel code after swapping operations A4 and A6.

VALUE	L1	L2	M3	A4	M5	A6
Allocation	LO	RO	RO	RO	LO	RO
Lifetime	13	7	6	6	6	4

Table 4: Allocation requirements of values for example loop after swapping operations A4 and A6.

have been scheduled in distinct clusters. If we where able to schedule both operations in the same cluster the value L1 would become local instead of global. This can be accomplished in two ways:

- By scheduling operations in the proper cluster so that the register requirements are reduced. This option has the problem that increases scheduler complexity, and requires the modification of the scheduling algorithm used.
- By swapping operations, after the scheduling phase, trying to assign each operation to the appropriate cluster (i.e. the cluster where the register requirements would be minimized). This option has the advantage that is simpler, and can be applyed independently of the scheduling algorithm.

We have chosen the second option. For instance consider that we swap operations A4 and A6 obtaining the kernel schedule of Figure 5. With this new schedule, operations M3 and A6 are in the left cluster, so value L1 is now left-only. In addition other values have changed from left-only to right-only and viceversa. Table 4 shows how values have to be allocated with the new schedule. The new schedule requires 19 left-only registers and 23 right-only registers resulting in a maximum of 23 registers in one cluster.

5 Experimental Analysis

5.1 Benchmarks

We have done experiments with inner loops belonging to the Perfect Club benchmark suite [10]. Loops have been selected with the following criteria: loops that perform floating-point calculations (because this work has been oriented to floating-point intensive applications), and that are composed of one basic block (because modulo scheduling is only applicable to loops with one basic block). We have not measured loops with conditionals in their body even though they can be converted to one basic block using IF-conversion. Almost 800 loops, accounting for the 57% of execution time of the whole Perfect Club[6], have been scheduled.

In order to obtain the dependence graph of the loops of the Perfect Club, the programs have been translated to optimized assembler code for the R3000 processor. Afterwards we developed a custom tool to extract dependence graphs from the innermost loops. The advantages of obtaining dependence graphs from optimized assembler code are that the compiler has performed some machine independent optimizations such as data reuse, common expression elimination, etc. Regretfully it has some drawbacks such as the presence of spill code. This problem is not very serious because the R3000 issues only one instruction each cycle and has short latencies, so it has few register requirements. We also eliminated this effect by detecting and eliminating spill loads and stores; the assembler code of the loops has been scanned looking for stores to the stack and posterior loads from the same position on the stack. When this pattern is found, the store and the load are removed from the dependence graph and dependences between the predecessor of the store and all the successors of the load are added.

5.2 Experiments

In our experiments we have considered machine configurations with an even number of functional units of each type in order to make comparisons between the unified/consistent register file with the non-consistent dual register file. In particular all the configurations have 2 adders and 2 multipliers and 2 load/store units, that is, each cluster has 1 adder, 1 multiplier, and 1 load/store unit. For the floating-point functional units we have experimented with two distinct latencies, 3 and 6. A latency of 1 cycle for loads and stores has been chosen because we assume a decoupled architecture; even though, this assumption can also be extended to a perfect cache which does not cause cache misses. All functional units are fully pipelined. The adders perform floating-point additions, subtractions, and type conversions between integer and floating point operands. The multipliers perform floating-point multiplications and divisions. Divisions execute with the same latency as multiplications.

Two kind of experiments have been performed. First of all we have measured the registers required by each configuration and then we have measured how these requirements affect performance. For the experiments we have considered the following models:

Ideal: it corresponds to an ideal machine with an infinite number of registers. We use this model to calculate an upper bound for performance.

Unified: it corresponds to a traditional unified register file and to a *consistent dual register file* organization.

Partitioned: it corresponds to a *non-consistent dual register file* without swapping of operations .

[6]Executed on a scalar processor of the CONVEX C3480 System, and timed with the loop performance analyzer CXpa.

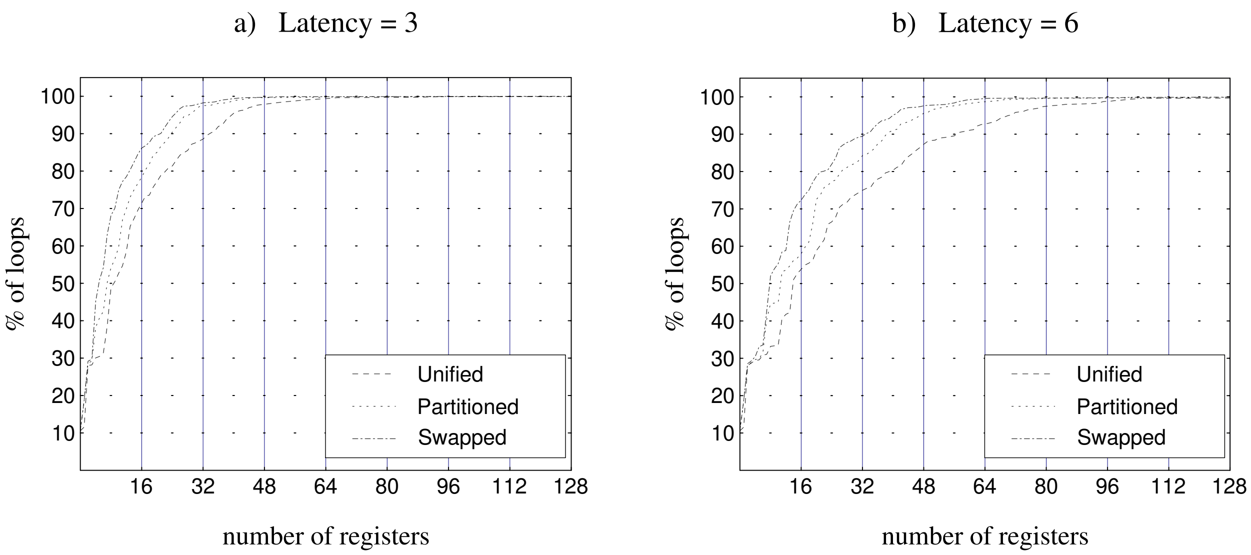

Figure 6: Cumulative distribution of loops.

Swapped: this model corresponds to the partitioned model plus operation swapping in order to reduce register requirements. After the scheduling phase, a greedy algorithm has been applyed to swap those operations that reduce the total number of registers required. The algorithm only swaps operations scheduled in the same cycle. In each step a pair of operations is selected to be swapped. The pair of operations selected to be swapped is the one that produces the highest reduction in the registers required. This step is repeated until there are no pairs that reduce registers required when they are swapped. Due to the cost involved to allocate registers, the registers required by each pair swapped is estimated by a lower bound on the registers required. This lower bound can be found by computing the maximum number of values that are alive at any cycle of the schedule.

5.3 Register Requirements

In order to measure register pressure, loops have been scheduled for the two configurations previously presented. The scheduling has been performed with the aim of achieving maximum performance, without regarding register usage. After the scheduling phase registers have been allocated trying to minimize the number of registers used, but with no restrictions in the number of registers available.

Figure 6 shows the static cumulative distribution of loops based on their register requirements for the two configurations and the three models studied. In this figure one can see the effect of the latency on the register requirements. For instance notice that configurations with latency 6 shows greater register requirements than the one corresponding to latency 3.

Notice that the "partitioned" model (i.e. partitioning the registers without swapping operations) produces a significant improvement on the registers required by loops. The "swapped" model also produces an improvement over the "partitioned" but it is not as noticeable as the previous improvement. In general the improvement of partitioning the register file is greater for configurations that require more registers.

The static plots shown in Figure 6 show that our approach can be applied to a high number of loops. To show that this approach is both applicable and useful, Figure 7 shows the dynamic cumulative distribution of the loops. In this figure the loops have been weighted by their estimated execution time. In order to measure the execution time we have measured the number of iterations each loop has been executed [7] times the II obtained once the loop has been modulo scheduled.

From this figure one can see that the loops that have higher register requirements represent an important part of the execution time of all the loops. The high dynamic register requirements of pipelined loops suggests that further research is required in the subject of register allocation and register organization.

Notice the big improvement in the registers required by the "partitioned" model when dynamic data is considered. It is also interesting to note the small difference on the registers required between the "partitioned" and the "swapped" models. This observation suggests that further improvements in the distribution algorithm would provide unappreciable improvements. Even thought the small difference between the plots of the "partitioned" and the "swapped" models, this difference can have an effect on performance as shown in the next subsection.

[7] Measured with the loop performance analyzer CONVEX CXpa.

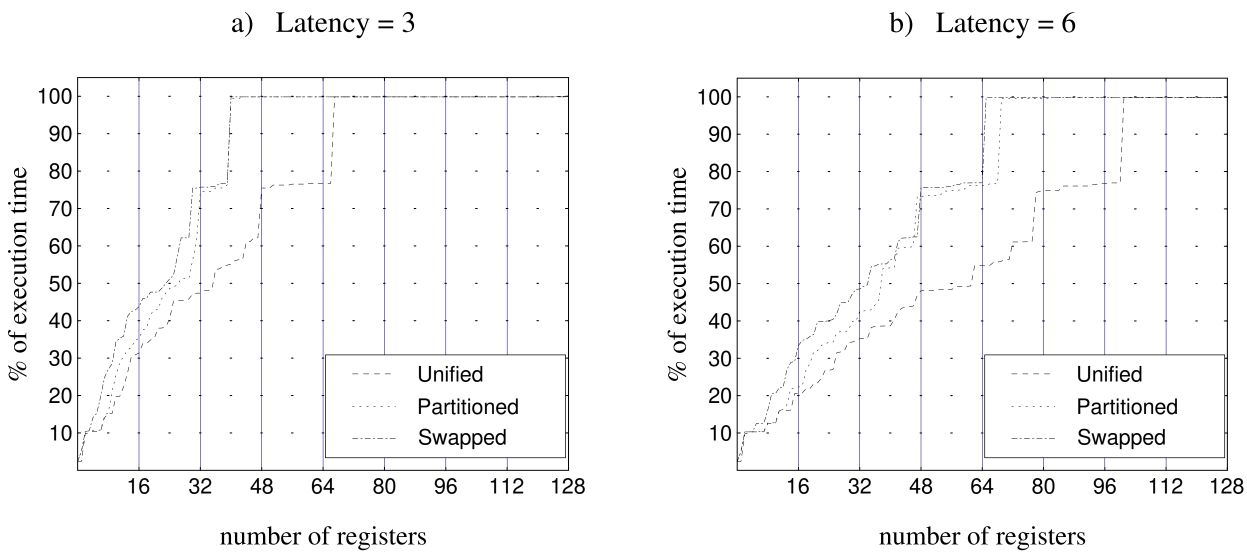

Figure 7: Cumulative distribution of cycles.

5.4 Limited Register Files

When there is a limited number of registers and the register allocator fails to find a solution requiring no more registers than those available, some additional actions must be taken. Different alternatives can be considered:

- One approach is to reschedule the loop with an increased II. The presumption is that register pressure is proportional to the number of concurrently executed iterations. This option would produce an extremely inefficient code, especially if the loops require many more registers than available.
- Another option is to split the loop into two or more loops, each one with fewer operations than the original one. In general, loops with fewer operations tend to require less registers, but it is not entirely clear that this is so; in fact the smaller loops will generally have a lower II, and the register pressure could even increase. In addition, memory traffic can increase if data generated by one piece of the loop is required by another piece of the original loop.
- The third option may be to select the appropriate lifetimes to spill, add the required spill code, perform modulo scheduling again, and repeat the register allocation. Unfortunately, in the new schedule it is possible that a different set of lifetimes might need to be spilled than those that have already been spilled. The addition of spill code will increase memory traffic and, potentially convert compute-bound and balanced loops to memory-bound loops or even increase the "memory-boundness" of memory-bound loops. When this situation occurs the new loop requires more cycles to be scheduled, and thus performance is reduced. Obviously even if no additional cycles are required for the II, the increase in memory traffic can also degrade performance.

In this section we study the effects of a limited register file when spill code is added. We will not consider the other two options to reduce register pressure. To insert spill code we have developed a "naive" spiller that works in the following way:

```
DO
   modulo scheduling
   register allocation
   IF registers needed > physical registers
      select a value to spill out
      modify the dependence graph
UNTIL registers needed <= physical registers
```

To select a value to spill out we have selected the value with the highest lifetime, which in general will free a higher number of registers. More research is required to develop better algorithms to spill registers in software pipelined loops, which would produce less performance degradation.

In order to study the effect on performance that a limited register file has, we have evaluated the effects on the density of memory traffic and the effects on the initiation interval (i.e. the effects on the maximum performance assuming a perfect memory system). The term density of memory traffic refers to the fraction of the bus bandwidth used on average each cycle, while the term memory traffic refers to the total number of memory accesses performed. The motivation of evaluating density of memory traffic, instead of the total memory traffic, is that an increase of the memory traffic can produce two negative effects. (1) An increase of the II, which we use to evaluate performance. (2)

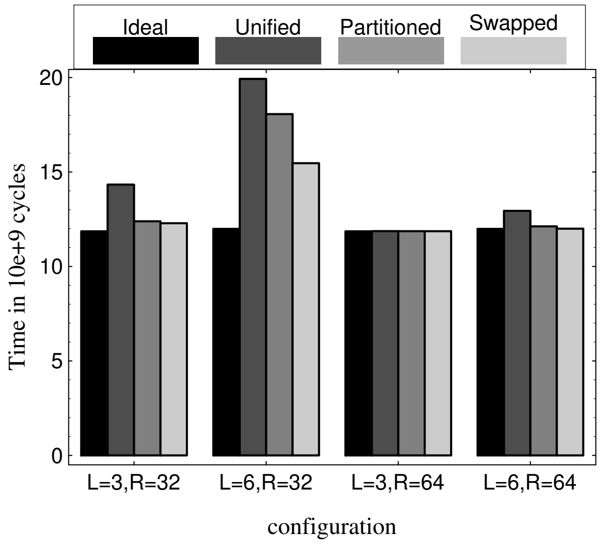

Figure 8: Performance for the four models considered with latency 3 and 6 and with 32 and 64 registers.

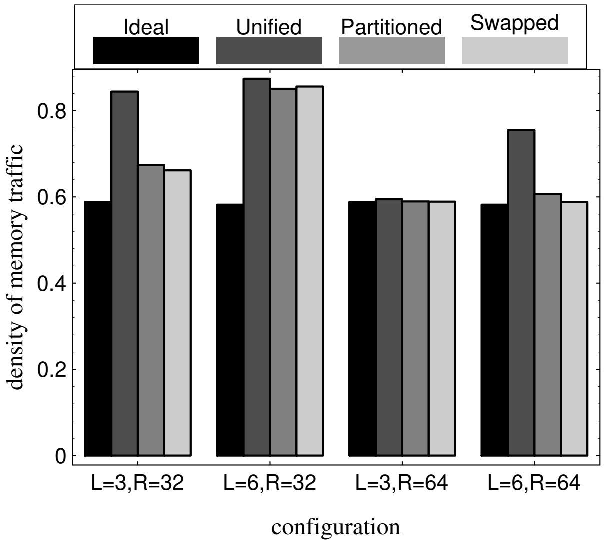

Figure 9: Density of memory traffic for the four models considered with latency 3 and 6 and with 32 and 64 registers.

An increase of the density of memory traffic, which in general will degrade performance of the memory system.

Figure 8 shows the performance for the two latencies and four models considered when the number of registers is limited to 32 and to 64.

When 64 registers are available, both models ("partitioned" and "swapped") almost achieve the same performance than the ideal case (i.e. infinite registers). On the contrary, the "unified" model has a loss of performance when a configuration with high latency is considered.

When 32 registers are available the "unified" model has a noticeable loss of performance specially for the configuration with high latency. On the other hand the "partitioned" and "swapped" models almost achieve the same performance than having infinite registers for the less aggressive configuration (i.e. latency 3). Notice that for the configuration where performance is highly degraded due to a lack of registers, the "swapped" model performs better than the "partitioned" model. This observation suggests that it is justified to use an expensive swapping algorithm if the execution time of the application can have a noticeable improvement.

Finally Figure 9 shows the density of memory traffic. The density of memory traffic corresponds to the dynamic average bus usage per cycle. With a perfect memory system it doesn't matter the density of memory traffic. If the effects of the memory where considered, the memory system could also degrade performance, and higher density of memory traffic could increase the possibility of degradation. Notice that, except for the configuration with latency 6 and 32 registers, the "partitioned" and the "swapped" models have less density of memory traffic than the "unified" model. For that case (i.e. L=6, R=32) the density of memory traffic is more or less the same for the three models due to the presence of a high amount of spill code.

6 Conclusions

As processors will increase the number of instructions executed per cycle, the number of registers required to exploit all the parallelism offered by the functional units will increase. At the same time the ports required to provide with data to these functional units will also increase. So alternative register file implementations will be required to support the bandwidth demands and the higher register requirements, without penalizing access time. We presented the *non-consistent register files* where local register instances are stored in the local register file of the cluster, while register instances used by both clusters are replicated on both clusters.

This proposal has been studied for a variety of machine configurations and tested with a collection of loops from the Perfect Club. The non-consistent dual register files significantly reduce the overall register requirements of loops. For example, about 10% more of the loops can be allocated without requiring spill code when a 32-register file is used. When dynamic execution is considered, those loops represent between 15% and 30% (depending on the configuration) of the execution time of all the loops.

We have also shown that the use of registers can be improved if an appropriate scheduling of operations to clusters is done. When swapping of operations is considered, the relative performance with respect to the unified register file can be improved up to 22% (depending on the configuration) and the density of memory traffic can be reduced in a 22%.

It would be interesting to consider better scheduling algorithms, but the high cost involved makes this approach unfeasible for a compiler (which is supposed to generate good code without requiring excessive processing time). On the other hand if the proposal of this paper were used for high-level synthesis purposes, it would be acceptable to consider better and expensive scheduling algorithms.

In addition, the register organization proposed is cheaper than doubling the number of registers (it requires less bits to specify the operands and requires less area), and does not penalize the access time to the register file. In some cases it is as effective as doubling the number of registers. Only for the configuration with latency 6 a *non-consistent dual register file* with 32 registers performs worse than a 64 registers unified register file (there is a degradation of performance of 13%).

References

[1] *DECchip 21064-AA Microprocessor Hardware Reference Manual.* Digital Equipment Corporation, 1992.

[2] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The Mips R4000 processor. *IEEE Micro*, 12(2):10–22, April 1992.

[3] Steven W. White and Sudhir Dhawan. POWER2: Next generation of the RISC System/6000 family. In *IBM RISC System/6000 Technology: Volume II.* IBM Corporation, 1993.

[4] Peter Yan-Tek Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[5] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[6] S. Jain. Circular scheduling: A new technique to perform software pipelining. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 219–228, June 1991.

[7] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*, pages 183–197, October 1981.

[8] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Register requirements of pipeline processors. In *Int. Conference on Supercomputing*, pages 260–246, July 1992.

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

[10] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, Center for Supercomputing Research and Development, November 1988.

[11] B.Ramakrishna Rau, David W.L. Yen, Wei Yen, and Ross A. Towle. The cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *IEEE Computer*, 22:12–35, January 1989.

[12] J.E. Smith. Decoupled access/execute computer architecture. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[13] J.C Dehnert, P.Y.T. Hsu, and J.P. Bratt. Overlapped loop support in the cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, 1989.

[14] G.H. Chaitin. Register allocation and spilling via graph coloring. In *Proc., ACM SIGPLAN Symp. on Compiler Construction*, pages 98–105, June 1982.

[15] B.R. Rau, M. Lee, P. Tirumalai, and P. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[16] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.

[17] Corinna Grace Lee. *Code Optimizers and Register Organizations for VectorArchitectures.* PhD thesis, University of California at Berkeley, 1984.

[18] A. Capitanio, N. Dutt, and Nicolau A. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *MICRO25*, pages 292–300, 1992.

[19] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *MICRO 25*, pages 236–245, December 1992.

[20] R.M. Russel. The CRAY–1 computer system. *Comunications of the ACM*, 21:63–72, January 1978.

[21] Special issue. *IBM Journal of Research and Development*, 34, January 1990.

[22] J. Llosa, M. Valero, J. Fortes, and E. Ayguade. Using Sacks to organize register files in VLIW machines. In *CONPAR 94 - VAPP VI*, september 1994.