Using Sacks to Organize Registers in VLIW Machines *

Josep Llosa, Mateo Valero, Jose A.B. Fortes** and Eduard Ayguade

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya.
c/ Gran Capità, Mòdul D6, 08071 Barcelona, SPAIN
e-mail: josepll@ac.upc.es

Abstract. This paper analyses the register requirements of software pipelined inner loops. When the number of functional units and/or the number of stages of individual functional units is increased, the number of registers required may be prohibitive in chip area and cycle time.
We characterize lifetime of values in pipelined loops with their **loop register locality** (LRL). Based on this characteristic, we propose a new organization of the register file in order not to affect cycle time and also reduce area, while increasing the number of registers. This can be useful to minimize the frequency of spill at a reasonable cost. The spill code can increase the minimum initiation interval and decrease loop performance. This new organization consists of a small high bandwidth multiported register file and a low bandwidth port-limited register file called *sack*. A mechanism to assign values to the *sack* is presented. We demonstrate the effectiveness of our approach by experimenting with a collection of loops from the Perfect Club benchmark suite. Experiments in order to find the optimal number of registers into the *sack* have been done. We also measured the effect of the spill code on loop performance.

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 which 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 papers that propose and evaluate different heuristic strategies to perform software pipelining [6][7]. Software pipelining techniques are applicable to RISC, CISC, superscalar, superpipelined, and VLIW processors, and they are useful whenever a processor offers parallelism either by having pipelined functional units or by allowing multiple operations to be issued per cycle.

* 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).

** School of Electrical Engineering, Purdue University.

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]. Usually, registers are organized in a multiported register file. If this multiported register file has a large number of registers and a large number of ports it can be expensive and increase cycle time reducing performance. If the number of registers required is bigger than the available number of registers, spill code has to be introduced to reduce register usage. This spill code can also reduce performance.

In this paper register requirements of pipelined floating-point intensive loops are studied as a function of the latency and the number of functional units (Section 3). We will conclude that the number of registers required to effectively exploit the available parallelism is high. We characterize the storage requirements of values and, based on this storage requirements define the **loop register locality** (LRL) (Section 4). Based on LRL a new register organization is proposed and evaluated. This register organization makes use of a port-limited register file with only one read port and one write port, which we call *sack* (Section 5). The main advantages of using *sacks* are their low cost (in area) and their fast access time compared to a multiported register file. *Sacks* can reduce the number of registers required in the main register file by holding those values that have high storage requirements in the conventional organization. Section 6 describes the experiments performed and shows how the proposed organization can improve performance by allowing fast access time to registers while reducing spill code. Finally, Section 7 states our conclusions.

2 Architecture Overview

In this section we describe the main parameters of the target machine of interest and a brief overview of certain terms associated with modulo scheduling, and hardware support for software pipelining. The target machine model we are dealing with is a VLIW floating-point processor, similar in some aspects to Cydrome's Cydra 5 [9]. This processor could be used as the data processor of a decoupled architecture [10] or as the floating-point part of a general purpose VLIW. Architectural support for software pipelined loops without code replication is assumed [11]. Table 1 shows the main characteristics of our base architecture.

Hardware support	Rotating register files Predicated execution
Operation types	Only floating-point operations
Memory ports	1 store and 2 load ports
Memory latencies	1 cycle
Code generation	Modulo scheduling
Register allocation	Wands Only strategy First Fit allocation schema

Table 1. Characteristics of the architecture used in the experiments.

Functional Units: Our base machine has 2 load memory ports and one store memory port. It also has a variable number of multiplier-adder pairs of variable latency depending on the configuration. The load and store latency of 1 cycle was chosen because our work is targeted to decoupled architectures. This latency also corresponds to that of loads and stores that do not cause cache misses. The adders perform floating-point additions, subtractions and type conversions between integer and floating operands. The multipliers perform floating-point multiplications and divisions. All functional units are fully pipelined. We assume that divisions are executed with the same latency and issue rate as multiplications. While this may not be very realistic, it does not distort our results significantly due to the low number of divisions present in the inner loops we experimented with.

Configurations used in the experiments: Let PxLy denote a configuration with x adder-multiplier pairs of latency y. In particular four configurations are used along the paper. Configuration P1L3 corresponds to a moderately pipelined/parallel design, while configurations P2L3 and P1L6 correspond to designs with aggressive parallelism and aggressive pipelining, respectively. Both techniques try to increase performance, the first by replicating functional units and the second by reducing cycle time. Finally configuration P2L6 corresponds to a design that combines both, aggressive parallelism and aggressive pipelining. Current and future advanced microprocessor architectures either have or can be expected to have similar configurations.

Modulo Scheduling is a scheduling technique which attempts to maximize resource utilization by overlapping the execution of successive iterations of a loop so that the most highly used resource within the loop is continuously busy. The number of instruction issue cycles between the initiation of successive iterations in a modulo scheduled loop is termed the *initiation interval (II)*. The schedule of an iteration can be viewed as a N-stage pipeline consisting of II cycles each *loop pipestage*. The number of *loop pipestages* in one iteration is termed the *stagecount*.

Rotating Register Files: When modulo-scheduling a loop, it is quite frequent for the result of an operation to be alive for more than II cycles, thus overlapping different (consecutive) outputs of the same operation. A rotating register file can solve this problem without replicating code. Values generated by an operation in one iteration will co-exist with the corresponding values in successive iterations. New values are written to successive physical cells in the rotating register file without interfering with previously generated values.

Register Allocation: Once a loop has been scheduled, register allocation determines its register requirements. How to allocate registers for modulo-scheduled loops is beyond the scope of this paper. For an extensive discussion of the problem see [12]. 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 registers. For this strategy all the allocation schemas have similar results, but First Fit has been selected due to its simplicity.

This paper concentrates only on floating-point loop variants, because the number of registers required to store loop invariants remain constant independently of the architecture. The addresses and integer values are assumed to be stored in the address processor of the decoupled architecture or in the case of a conventional VLIW, in the integer unit. So, from here on "registers" refers to the rotating register file used to store the loop variants. The techniques proposed to efficiently organize the rotating register file can also be used in machines without a rotating register file, and for loop invariants; the focus is on loop variants because their register requirements increase with latency (stages) of functional units and number of functional units, as shown in the next section.

3 Register Requirements of Pipelined Loops

In this work we have experimented with nearly 800 loops belonging to the Perfect Club benchmark suite [13] in order to measure the register pressure of real loops, as the number and latency of functional units increase. Of all the innermost loops of the Perfect Club only those loops that perform floating-point calculations have been selected, because this work has been oriented to numerical applications. Of these loops we have scheduled those loops composed of one basic block; that is loops without conditional branching and without calls to subroutines. The execution time of the selected loops accounts for 57% of the execution time of the whole Perfect Club[3].

The loops have been modulo-scheduled for the four configurations of Section 2 followed by register allocation. To measure register pressure, the register allocation strategy assumed that an infinite number of registers were available. A value is considered to be alive (a register is required to store the value) from the cycle the producer starts, to the cycle the last reading instruction ends. Table 2 shows the percentage of loops that need no more than 16, 32 and 64 registers, the percentage of loops that need more than 64 registers and the maximum number of registers required by the selected loops. Data in the column "% of loops" represent the static measurements of loops (number of loops). Data in the column "% of cycles" represent the dynamic measurements of loops (that is, instead of counting the number of loops, we have measured the number of cycles spent by them). In order to measure the number of cycles we have computed the total number of iterations each loop has been executed, and we have multiplied this value by the II obtained once the loop is scheduled.

Table 2 shows that, when the architecture becomes more parallel and its functional units more deeply pipelined, the percentage of loops that may require a large number of registers increases. Seemingly, an increase in the latency has more effect on the number of registers than an increase on the number of functional units. This is mostly due to the fact that all configurations have the same memory bandwidth. Thus, loops that are memory-bound on a one pair multiplier-adder configuration, cannot use further parallelism when they are scheduled on a two pair multiplier-adder configuration. If this happens the

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

loop does not require more registers than if scheduled on a one pair configuration. Only those loops that can exploit more parallelism than with a one pair configuration increase their register requirements. It is curious to note that the percentage of cycles spent in loops with few registers increases when the number of functional units increases. This is due to the fact that most of the loops that require few registers are memory-bound and do not improve their II when scheduled with more functional units.

	% of Loops				% of Cycles				
	16	32	64	>64	16	32	64	>64	MAX
P1L3	63.1%	95.2%	99.7%	0.3%	14.5%	66.4%	99.8%	0.2%	148
P2L3	60.4%	84.9%	99.2%	0.8%	22.9%	41.1%	74.6%	25.4%	178
P1L6	43.8%	72.2%	97.6%	2.4%	8.6%	21.1%	68.4%	31.6%	185
P2L6	43.0%	65.4%	89.4%	10.6%	13.9%	24.4%	50.9%	49.1%	260

Table 2. Percentage of loops that can be allocated without spilling. The last column is the maximum number of registers required by a loop.

In addition to the effect of pipelining and replication of functional units on the requirements of registers, the compiler can apply transformations such as blocking to improve performance of memory-bound loops. These techniques improve performance at the expense of an increment in the register pressure. So, it would be desirable to have as many registers as possible in order to generate efficient code. Having a large number of registers implies that the register allocator would find a feasible solution requiring no more registers than available for a large number of loops. When this is not possible, different alternatives can be used to deal with the insufficient number of registers. Increase the II, which would produce extremely inneficient code, especially if the loops require many more registers than available. Or introduce spill code, which would increase memory traffic, and convert compute-bound and balanced loops to memory-bound loops, or increase the traffic requirements of memory-bound loops. When a loop becomes memory-bound or increases its memory-boundness it requires more cycles to be scheduled.

Unfortunately it is not always possible to have as many registers as we need because having a large number of registers implies more silicon area devoted to registers and greater access time, which, in its turn, can affect the cycle time if the register file is in the critical path.

4 Loop Register Locality

In this section we characterize values used in a software pipelined loop by their **storage requirements** and by their **loop register locality**. Before defining these concepts, we present a simple loop, perform modulo scheduling and analyze lifetime for all values. From this analysis, we will derive these two key characteristics.

Consider the loop of Figure 1a whose data dependence graph is shown in Figure 1b . Let us assume a machine with the configuration shown in Figure 1c. When Modulo Scheduling is used, the schedule shown in Figure 2 is obtained.

a)

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

c)

Functional Unit	Number	Stages
Adders	1	4
Multipliers	1	4
Load Ports	2	1
Store ports	1	1

b)

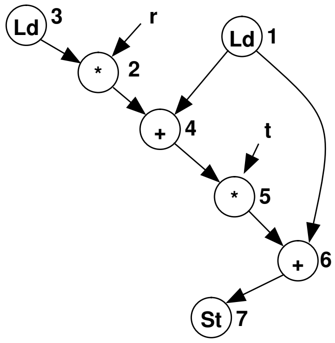

Fig. 1. a) Fortran loop, b) data dependence graph of the loop, and c) machine configuration for the example.

Note that the minimum number of additions and multiplications required to perform the calculations in the loop is two. This means that the saturated resources are the adder and the multiplier. The minimum instruction 'packed' size to execute this loop is therefore two. The schedule is partitioned into 10 pipestages, each of which is 2 cycles in duration. Ten concurrent iterations are required to saturate the multiplier and the adder. Notice that multiplication 5 has been placed in cycle 10 instead of cycle 9. This ordering is required because only one instruction can be dispatched per cycle per functional unit, and cycle 9 (second cycle on the compressed schedule) is used by multiplication 2.

		Functional Unit					Value					
Pipestage	Time	Mult	Add	Load	Load	Store	1	2	3	4	5	6
1	0			Ld 1	Ld 3		w		w			
	1	$*$ 2					x		r			
2	2						x					
	3						x					
3	4						x	w				
	5		+ 4				r	r				
4	6						x					
	7						x					
5	8						x			w		
	9						x			x		
6	10	$*$ 5					x			r		
	11						x					
7	12						x					
	13						x				w	
8	14		+ 6				r				r	
	15											
9	16											
	17											w
10	18					St 7						r
	19											

Fig. 2. Scheduling for the loop of Figure 1a, and value lifetime for each node: w means that the value is written in this cycle, r means that this value is read in this cycle and x means that the value is alive in this cycle.

Assume that registers are written in the last cycle of the operation and that they are read in the first cycle of the consumer operation. The lifetime of a value has been computed, for this example, from the cycle when the value is written to the cycle when the last read of this value is performed. The values generated by each operation are alive during the cycles shown in Figure 2. Each iteration (a new iteration is initiated every II cycles) of the loop generates 6 new values that overlap with those produced in previous iterations because some of them have lifetimes greater than the II. For example in the previous scheduling there are up to 8 alive results of operation 1 overlapping simultaneously. Overlapping lifetimes each II cycles (II = 2 in this example) 14 registers are required at least to allocate those 6 values . Eight of these registers are used to allocate the results generated by operation 1.

We define the **storage requirement** (SR) of a value as the minimum number of physical registers required to store all the concurrent results of this value, that is, the lifetime of the value divided by the II of the schedule. In our example value 1 has a SR of 7.5, this means that all the simultaneously alive results of value 1 use 7 registers during II cycles and another register during 0.5II cycles.

We define the **loop register locality** (LRL) of a value as the number of times a value is accessed (either for read or write) every II cycles divided by the SR of this value. The concept of LRL relates to how often the physical registers are accessed, instead of how often the values are accessed. As an example consider value 1 (LRL = 0.4) and value 2 (LRL = 2). Value 1 is written once and read twice so it is accessed three times each iteration. Even though value 1 has more temporal locality than value 2 (written once and read once each iteration), it has less LRL than value 2. Value 2 requires only one register to be allocated which is accessed two times per iteration, while value 1 requires 8 physical registers [4] to be stored each one of them is accessed 3 times every 8 iterations.

5 A Register Organization Based on LRL

Based on their LRL, values of software pipelined loops can be classified into low locality values and high locality values. This difference can be used to split the register file into two register files: a multiported register file with no bandwidth limitation and a bandwidth-limited register file with only one write port and one read port named a *sack*. We propose to use the *sack* to store the values that have low LRL and to use the multiported register file to store the values with high LRL. The main advantage is that the *sack* is cheaper (in area) and faster than the multiported register file, so it can contain a high number of physical registers.

In the previous example and from lifetimes of Figure 2 we can generate a compressed reservation table of reads and writes of length II for each value (as shown in Figure 3a). This table records the cycles where the values are read and written and the information in this table can be used to assign values to the *sack*. Given that an iteration is initiated every II cycles and that there are several overlapping iterations executing concurrently, it is possible for a value to be read

[4] Actually it requires 7.5 registers and the register allocator can use the remainder half register to allocate other values.

several times, corresponding to different iterations, in a cycle. So the entries in the read reservation table can have an arbitrary number, between 0 and the maximum number of reads that the functional units are able to perform in one cycle. For example, if the previous loop was scheduled for a configuration with two adders and two multipliers, we would obtain a compressed scheduling where the additions 4 and 6 would be executed in the same cycle of the compressed schedule and, therefore there would be two reads of value 1 corresponding to different iterations of the uncompressed scheduling.

Assume that we have an organization with one multiported register file and one *sack*. For example if we decide to allocate value 3 in the *sack* we cannot allocate simultaneously value 2 because both values are read and written at the same time. But we can allocate value 6 because it is read and written at different cycles than value 3. If we allocate value 1 in the *sack* we cannot allocate any other value because value 1 saturates the read port of the *sack*. In this last case, that is, allocating value 1 in the *sack* and the remaining values in the multiported register file, we need 6 (expensive) registers in the multiported register and 8 (cheap) registers in the *sack*.

If we use a *sack* we have to solve two assignment problems: assigning values to registers and assigning values to the *sack* so that no conflicts occur when transferring values. The first of these problems is the traditional problem of register assignment and, for modulo scheduled loops, has been studied in [12]. In the rest of this section, we describe the algorithm we propose to assign values to the *sack*. A set of values can be assigned to a *sack* if their reservation tables are fully compatible; that is, no more than one read and one write access is allowed in a cycle. Thus, values whose reservation table has more than one read in one cycle cannot be allocated to a *sack*. These values are forced to be allocated in the multiported register file where several simultaneous reads and writes are allowed.

a)

	Cycle	Value					
		1	2	3	4	5	6
Reads	1	1	0	0	1	1	1
	2	1	1	1	0	0	0
Writes	3	1	1	1	1	0	0
	4	0	0	0	0	0	1

b)

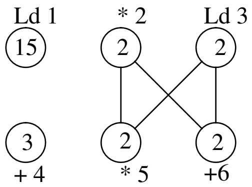

Fig. 3. a) Compressed reservation table of reads and writes for the scheduled loop in Figure 1. b) Compatibility graph for the vaues in the loop.

Compatibility Graphs: As the registers in the *sack* are cheaper than the registers in the multiported register file, it would be interesting to assign as many registers as possible to the *sack*, in order to have a multiported register file as small as possible. We estimated that by assigning those values with the largest lifetimes to the *sack*, the objective of minimizing the number of multiported registers would be nearly accomplished. So, the objective is to find a set of values which minimizes the multiported registers while maintaining compatibility between all the reservation tables of those values. To find this set of values, we construct the *compatibility graph* of the loop, which indicates the benefits of assigning a value to a *sack* and when an assignment will not result in conflict. In a *compatibility graph*, a vertex is a value, and an edge between two vertices represents no conflict between two corresponding values. Each vertex of the *compatibility graph* has an associated weight, which is the lifetime of the value represented by the vertex. Figure 3b shows the *compatibility graph* for the example used in the previous Section.

Once the *compatibility graph* has been constructed, the problem is reduced to finding a fully-connected subgraph (this guarantees that the selected nodes are fully compatible) which, in addition, maximizes the sum of weights associated to the nodes of the subgraph. We call this subgraph the *maximum subgraph*. In our example there are several subgraphs whose nodes are fully connected but only the subgraph that contains value 1 maximizes the sum of weights. Unfortunately computing the *maximum subgraph* is an NP-hard[14] problem. As a result, we use a heuristic to maximize the number of registers assigned to the *sack* with no guarantee that the maximum has been found. The algorithm used to assign values to a *sack* is:

```
MaxWeight := 0;
FOR each vertex V of the compatibility_graph DO
  Subgraph := V;
  WHILE there are vertex connected to all vertex in Subgraph DO
    Subgraph := Subgraph + MAX(of all vertex connected to Subgraph)
  endWHILE
  IF weight(Subgraph) > MaxWeight  THEN
    MaxWeight := weight(Subgraph);
    MaxSubgraph := Subgraph ;
  endIF
endFOR
```

This algorithm computes a greedy subgraph connected to each node of the *compatibility graph* and finally selects the best of these subgraphs. It has a complexity of $O(n^2)$ where n is the number of vertices in the compatibility graph.

6 Experimental results

In this section, we evaluate how effectively values can be assigned to the *sack*, that is how many registers we need in the multiported register file in the case that a *sack* can be used to allocate some of the values. We also evaluate the number of registers required in the *sack* that is, the ideal number of registers that can be effectively used. The experiments have been done with the same loops used in Section 3. The same configurations have been used for the experiments.

Table 3 shows the percentage of loops that can be allocated with 0, 16, 32, 64, and 128 registers in the multiported register file and the percentage of cycles those loops represent. For example, for the configuration P1L3 using a *sack* and a multiported register file with 16 registers, the algorithm schedules 86% of the loops without spilling (representing 37% of execution time). The difference between static and dynamic measurements is due to the fact that a few big loops (that require a big number of registers) are responsible for most of the time spent by all the loops.

Notice in Table 3 that the same benefit (in terms of percentage of loops that can be fully allocated) is nearly obtained by either doubling the number of registers in the multiported register file or by adding a *sack*. For example, for the configuration P1L3 the same dynamic results are obtained by going from 32 to 64 registers than by adding a *sack*. The same happens for configurations P2L3 and P1L6 with 64 registers. In general it is more effective to increase the size of the multiported register file than to add a *sack* although it can be more expensive and increase cycle time.

We have also evaluated the maximum number of registers that can be allocated to the *sack*. Table 4 shows the percentage of loops that do not require more than 8, 16, 24, 32 and 40 registers in the *sack* and the percentage of dynamic cycles these loops represents. For this experiment we tried to allocate as many registers as possible in the sack. From this table it can be seen that no more than 32 registers are required in a *sack* for all the configurations. This can be explained by the fact that a *sack* has limited bandwidth and this limits the number of registers that can be effectively accessed in the *sack*.

Config	Sack	% of Loops 0	16	32	64	128	% of Cycles 0	16	32	64	128
P1L3	no	11%	64%	95%	100%	100%	2%	15%	66%	100%	100%
	yes	21%	86%	100%	100%	100%	5%	37%	100%	100%	100%
P2L3	no	11%	61%	85%	99%	100%	3%	23%	41%	75%	100%
	yes	21%	76%	94%	100%	100%	8%	31%	57%	100%	100%
P1L6	no	11%	45%	72%	98%	100%	2%	9%	21%	68%	100%
	yes	21%	63%	93%	100%	100%	5%	15%	67%	100%	100%
P2L6	no	11%	44%	65%	89%	100%	3%	14%	24%	51%	100%
	yes	21%	57%	80%	97%	100%	8%	21%	34%	73%	100%

Table 3. Percentage of loops that can be allocated with and without a *sack*

Config	%of Loops 8	16	24	32	40	%of Cycles 8	16	24	32	40
P1L3	79%	99%	99%	100%	100%	25%	68%	69%	100%	100%
P2L3	83%	99%	99%	100%	100%	40%	73%	75%	100%	100%
P1L6	59%	85%	97%	99%	100%	15%	31%	68%	100%	100%
P2L6	60%	85%	97%	99%	100%	24%	42%	73%	99%	100%

Table 4. Percentage of loops that can be allocated with *sacks* of 8, 16, 24, 32 and 40 registers.

Notice in Table 3 that with a 64 multiported register file and for the configuration P2L6, even though the algorithm allocated 97% of the loops, these loops represent 73% of the number of cycles. In [15] we evaluate the efficiency of using more than one *sack* and present a naive algorithm to allocate values to them. For instance, using two *sacks* with no more than 32 registers each one and a multiported register file with 64 registers we can obtain similar results than when a 128 register file is used.

Finally, to show the importance of reducing spill code we have measured its effects on performance when the number of registers is limited. Spill code has two negative effects. It increases the memory traffic with a possible loss of performance on the memory system. And, as stated in Section 3, some loops can increase their II because the loop becomes memory-bound. We have studied the direct effect on the II by adding spill on those loops that require more than 32 registers. Table 5 shows the relative performance of each configuration compared to the ideal configuration with infinite registers. Relative performance has been measured as the total cycles spent by all loops with spill for configuration PxLy divided by total cycles spent by all loops for configuration PxLy with infinite registers. For instance the introduction of spill code in the configuration P2L3 with 32 registers reduced total performance of the selected loops to 0.8668 compared to the same configuration with an unlimited number of registers. Note that the spill code has a considerable influence in performance, but for these machine configurations for which a *sack* is enough, the loss of performance is insignificant. An extended study of the performance degradation due to spill code can be found in [16].

	32 reg	+1 sack	64 reg
P1L3	0.9964	0.9998	0.9999
P2L3	0.8668	0.9835	0.9994
P1L6	0.9332	0.9984	0.9992
P2L6	0.6867	0.8165	0.9420

Table 5. Performance of each configuration (relative to having infinite registers) when spill code is added for the cases of having 32 registers, 32 registers and a *sack*, and 64 registers.

7 Conclusions

In this paper, we have analysed how register requirements of software pipelined loops are influenced by the number and stages of functional units and measured these requirements for various processor configurations. We have shown that these register requirements are considerable and that loops that require a high number of registers have an important effect on the execution time of floating-point intensive applications. This observation suggests that future architectures should be provided with a high number of registers (64 or more).

We also introduced a property of lifetime of values termed **loop register locality** and used it to propose a possible approach to organize the register file and an algorithm to assign values. The proposed organization consists of a small multiported register file, plus a port-limited register file called a *sack*. This organization has proved to be effective although it is not as effective as doubling the capacity of the multiported register file. As future work we plan to develop an algorithm to be integrated into a compiler in order to allocate registers, assign values to several *sacks*, and add spill when necessary taking into account the number of registers, the number of *sacks* and the number of registers in the *sacks*.

An experiment with limited register files has been done in order to measure the effects of spill code. We have shown that spill has a significant effect over performance (even though we have not measured its influence on memory performance), which suggests the use of large register files (to avoid spilling) and also that more research effort is needed to develop algorithms to generate better spill code for pipelined loops. In [15] we have done an exhaustive study of the *sacks* organization and how spill affects the II of loops and the memory traffic for a variety of organizations.

References

1. DECchip 21064-AA Microprocessor Hardware Reference Manual, *Digital Equipment Corporation*, 1992.
2. MIPS R4000 Microprocessor User's Manual, *Mips Computer Systems Inc.*, 1991.
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. Silicon graphics TFP micro-supercomputer chipset. In *Hot Chips V*, August 1993.
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. B.Ramakrishna Rau, David W.L. Yen, Wei Yen, and Ross A. Towle. The cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. In *IEEE Computer Vol. 22*, pages 12–35, January 1989.
10. J.E. Smith. Decoupled access/execute computer architecture. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.
11. 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.
12. 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.
13. 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.
14. Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness.* W.H.Freeman and Company, 1979.
15. J. Llosa and al. Using Sacks to organize register files in VLIW machines. Technical Report RR-94/06, Universitat Politecnica de Catalunya, 1994.
16. J. Llosa, M. Valero, E. Ayguade, and J. Labarta. Register requirements of pipelined loop and their effect on performance. To appear in *2nd Int. Workshop on Massive Parallelism: Hardware Software and Applications*, October 1994.

This article was processed using the LaTeX macro package with LLNCS style