Two-level Hierarchical Register File Organization for VLIW Processors

Javier Zalamea, Josep Llosa, Eduard Ayguadé and Mateo Valero *
Departament d'Arquitectura de Computadors (UPC)
Universitat Politècnica de Catalunya
{jzalamea,josepll,eduard,mateo}@ac.upc.es

Abstract

High-performance microprocessors are currently designed to exploit the inherent instruction level parallelism (ILP) available in most applications. The techniques used in their design and the aggressive scheduling techniques used to exploit this ILP tend to increase the register requirements of the loops. If more registers than those available in the architecture are required, some actions (such as spill code insertion) have to be applied to reduce this pressure, at the expense of some performance degradation. This degradation could be avoided if a high–capacity register file were included without causing a negative impact on the cycle time of the processor.

In this paper we propose a two-level hierarchical register file organization for VLIW architectures that combines high capacity and low access time. For the configuration proposed in this paper, the new organization achieves a speed–up of 10–14% over a monolithic organization with 64 registers; it is obtained with a 43% (40%) reduction in area (peak power dissipation). Compared to a monolithic file with 32 registers, the speed–up is as much as 38% with just a 14% (4%) increase in area (peak power dissipation).

1. Introduction

Current high-performance microprocessors use hardware and software techniques to exploit the instruction level parallelism (ILP) available in applications. Their architecture makes use of deep pipelines in an attempt to reduce the cycle time and simultaneous issue of operations in order to increase the number of instructions executed per cycle. It is expected that future designs will make extensive use of both techniques. Therefore, new processor organizations and compiler techniques are required to effectively exploit this potential parallelism.

*This work has been supported by the Ministry of Education of Spain under contract TIC 98/511 and by CEPBA (European Center for Parallelism of Barcelona). Javier Zalamea is granted by the Agencia Española de Cooperación Internacional.

The proper scheduling of instructions plays a critical role in the final performance. This scheduling is done at run–time in out–of–order superscalar processors (with the aid of the compiler which performs extensive code scheduling to facilitate the dynamic detection of parallelism). However, in Very Long Instruction Word (VLIW) architectures the scheduling of instructions is done at compilation time. The static nature of VLIW schedules requires good compilation techniques that effectively exploit the ILP available in programs [4, 12, 20].

Loops are the main time consuming part of numerical programs. Software pipelining [5, 14] is a loop scheduling technique that extracts parallelism from loops by overlapping operations from various consecutive iterations. Modulo scheduling [8, 22] is a class of software pipelining algorithms which has been incorporated in many production compilers. In a modulo scheduled loop, the *Initiation Interval* (*II*) is the number of cycles between the initiation of successive iterations. For a loop, the lower the *II* the higher the number of operations executed per cycle. For example, Figure 1 shows the average number of floating point computations performed per cycle[1] for different processor configurations *GPxMy-REGz* (*x* being the number of general–purpose floating–point functional units, *y* the number of memory ports, and *z* the number of registers in the register file). Notice that, in general, increasing the number of resources results in an increase in the performance achieved. For each loop and processor configuration, the *II* is bounded either by recurrences in the dependence graph or by resource constraints in the target architecture. For instance, increasing the number of functional units by a given amount may not result in the same increase of performance because of the loops which are limited by either recurrences or by other resources.

The drawback of aggressive scheduling techniques such as modulo scheduling is their high register requirements

[1] Assuming the experimental workbench (set of loops, modulo scheduler, ...) described in Section 2.

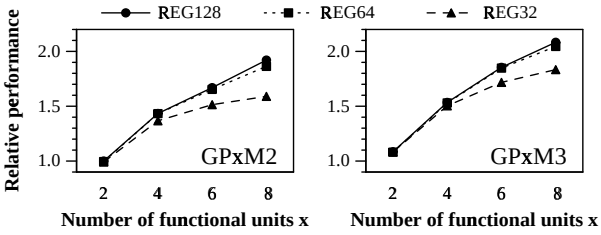

Figure 1. Average number of floating point computations performed per cycle for processor configuration *GPxMy-REGz*, relative to *GP2M2-REG32*.

[17] compared to less aggressive and less effective scheduling techniques. In addition, the use of aggressive processor configurations tends to increase the number of registers required by software pipelined loops. For this reason, many proposals have focused on minimizing the register requirements of modulo scheduling [9, 11, 18]. However, despite these techniques, many registers are still required. For instance, for one of the above mentioned processor configurations (*GP6M2*), the dashed line in Figure 2 shows the percentage of loops that can be scheduled with a specific number of registers using a register–conscious modulo scheduler. The solid line in the same figure shows the percentage of cycles spent in the execution of these loops. Notice that although less than 15% of the loops require more than 32 registers (and even less require more than 64 registers), they represent close to 40% of the total execution time. Other optimizations applied to loops (such as unrolling, common subexpression elimination, back substitution, ... [17]), techniques oriented towards hiding the negative effects of cache misses (such as prefetching [2] or blocking), breaking the data dependences (such as data speculation) or breaking the control dependence flow (predication, control speculation) increase even more the register requirements.

When a loop requires more registers than available, register pressure must be decreased by either increasing the *II* or by adding spill code (i.e. temporarily storing values in memory and freeing, for several cycles, the registers used). These two alternatives degrade performance at the expense of alleviating the high register demand. The evaluation performed in [16] shows that reducing the execution rate tends to generate worse schedules than spilling variables. New heuristics for register spilling have been proposed and proven to be very effective [28]. In any case, the performance degradation is still significant and could be avoided

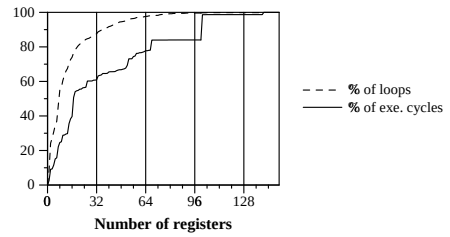

Figure 2. Register requirements for *GP6M2*.

if additional capacity were provided in the register file.

The organization and management of the register file has been a subject of research in the past. The main idea behind all this research is to trade off aspects related to storage capacity, area, cycle time and power dissipation of the register file. The monolithic register file organization traditionally used in the design of microprocessors does not scale well when the register requirements and the number of ports required to access it are high. In this paper we present an alternative design for the register file of future aggressive VLIW processors that tries to combine high capacity and high number of ports with low access time. The higher capacity reduces spill code and allows the application of aggressive software prefetching techniques.

This paper is organized as follows: Section 2 presents the framework used to evaluate the performance of our proposal. Section 3 studies the behavior of software pipelined loops for different configurations of the VLIW processor in which the number of registers in the register file, the memory bandwidth and the latency are varied. From this study, which includes the effect of the cycle time of the processor, the necessity for an alternative design of the register file is foreseen. Section 4 proposes the use of a two–level hierarchical organization; cycle time, area and peak power dissipation are the metrics used to compare the proposal with other monolithic register file organizations. Section 5 describes alternative designs proposed in the literature. Section 6 evaluates the proposal, assuming both an ideal memory system (always hits in the cache) and a real memory environment. Finally, Section 7 concludes the paper and presents some future work.

2. Evaluation framework

The proposal presented in this paper is validated using the framework described in this section. A workbench composed of all the loops from the Perfect Club benchmark [1] that are suitable for software pipelining is considered representative of the loops in numerical applications. A total of 1258 loops, that represent about 80% of the total execution time of the benchmark have been scheduled using HRMS (Hypernode Reduction Modulo Scheduling) [18], a register–conscious pipeliner. HRMS generates near–optimal schedules with minimum register requirements. If the schedule generated by HRMS does not fit into the available number of registers, an iterative process is started in which the register requirements are progressively reduced. This process combines the two techniques mentioned above (i.e. increase of the *II* and the insertion of spill code) [28].

The evaluation framework includes a set of statically scheduled VLIW configurations *GPxMy–REGz* already defined as follows: x is the number of general–purpose floating–point functional units, y is the number of memory ports (number of load/store units) and z is the number of

registers in the register file. In all configurations, the latencies of operations performed in the functional units are: 4 cycles for addition and multiplication, 17 cycles for division and 30 cycles for square root. All operations are fully pipelined except for division and square root. In this paper, we focus our study and experimental evaluation on aggressive processor configurations which could be implemented in a near future and which result in a reasonable efficiency (see Figure 1): x=6, y=$\{2,3\}$ and z=$\{16,32,64,128\}$.

The memory is designed around a multi–ported memory system (y ports) with a *L1–cache* of 32 Kb and 32 bytes line size. The *L1–cache* is lockup–free and allows up to 8 pending memory accesses. Hit latency for load operations is 2 cycles. Write operations take one cycle to complete. Miss latency is assumed to be in the range 4–32 cycles.

The performance metrics used in the evaluation are: execution cycles (directly obtained from the *II* and the number of iterations of the loops), memory traffic (including spill code), and execution time (where the access time of the register file is considered). The evaluation also compares the area occupied by the register file organization. Aspects related to power dissipation are briefly considered in this paper. All figures are given relative to a baseline configuration *GP6M2–REG128*.

In order to estimate access time, area and power dissipation for the different register file configurations, we use the model described in [23] targeted at a CMOS process with a minimum drawn gate length of 0.18μm. These figures depend on the number of registers, the number of access ports (which is in turn determined by the number of functional units x and memory ports y), and the clock frequency. For instance, for *GP6M2–REGz*, the number of ports is: 14 read ports (2 for each functional unit and 2 for the memory) and 8 write ports (one for each functional unit and 2 for the memory). Since the register file is the main centralized structure in a VLIW architecture, it can be assumed that the cycle time of future processors will be determined by the access time to the register file. For instance, Table 1 shows the cycle time, area and peak power dissipation for the set of monolithic register file organizations considered in this paper. Cycle time is given in nanoseconds, area is given in millions of λ^2 and power is given in watts. Also, all are given relative to the baseline configuration.

	Cycle time	*Area*	*Power*
Reg	ηs (relat.)	$\lambda^2 * 10^6$ (relat.)	W (relat.)
GP6M2			
128	1.651 (1.000)	5.325 (1.000)	10.409 (1.00)
64	1.451 (0.879)	2.662 (0.500)	6.030 (0.58)
32	1.280 (0.775)	1.331 (0.250)	3.504 (0.34)
16	1.130 (0.685)	0.666 (0.125)	2.054 (0.20)
GP6M3			
128	1.687 (1.022)	6.193 (1.163)	12.435 (1.19)
64	1.479 (0.896)	3.097 (0.582)	7.222 (0.69)
32	1.303 (0.789)	1.548 (0.291)	4.203 (0.40)
16	1.149 (0.695)	0.774 (0.145)	2.466 (0.24)

Table 1. Cycle time, area and power dissipation for some register file configurations: absolute and relative to a baseline configuration *GP6M2–REG128*.

3. Register requirements for VLIW schedules

The register requirements of software pipelined loops increase with the aggressiveness of the processor configuration and the latencies of the functional units and memory used. Techniques that try to hide the long latency of cache misses (like prefetching) also increase the register requirements. In this section we analyze the behavior, in an ideal memory system, of the workload considered in this paper when varying the capacity of the register file, the number of ports to memory and the latency of memory.

3.1. Register file size

For a particular VLIW architecture, the number of registers available in the register file may constrain the maximum performance that software pipelined loops may attain. For a loop, if its register requirements are larger than the size of the register file, spill code is required in order to reduce this pressure. Spill code increases memory traffic and therefore may reduce performance (increase of the *II* due to a saturation of the memory ports).

Figure 3 shows the behavior of our benchmark set for the 4 register file configurations considered. When the size of the register file is increased, the number of cycles needed to execute the loop and the memory traffic are reduced. This is due to the reduction of the spill code. For instance, configuration *GP6M2–REG16* needs 1.6 times more cycles than *GP6M2–REG128*. Similarly, *GP6M2–REG16* generates about 2.3 times more memory traffic than *GP6M2–REG128*. However, the execution time plot (Figure 3.c) shows a direct trade–off between the register file size and the actual performance. This plot is obtained by multiplying the values in Figure 3.a by the cycle time of each register file (Table 1). Notice that *GP6M2–REG128* results in the best performance in terms of number of cycles. However, the high access time of its register file results in a clear degradation of the execution time. In terms of execution time, *GP6M2–REG32* performs best in our workbench.

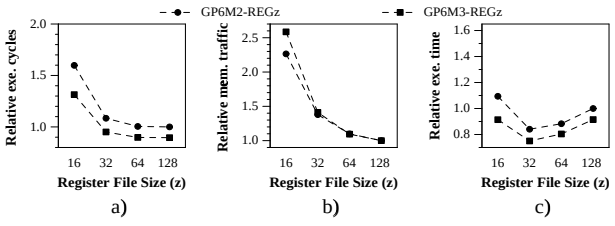

Figure 3. Behavior for several register file configurations relative to the baseline *GP6M2–REG128*.

3.2. Memory ports

Increasing the number of ports to memory makes the architecture more aggressive in terms of achievable ILP. This clearly benefits memory bounded loops (i.e. loops limited by the number of memory ports). Furthermore, having an additional memory port also gives more opportunities to schedule spill operations which are needed when the register requirements are higher than the actual register file size.

Having an additional memory port does not contribute to a high increase in the access time of the register file. As shown in Table 1, the gap between the two register file configurations with the same size is not significant (close to 2%). Moreover, the additional memory port complicates the design of the memory hierarchy (with an increase in the memory latency, the area and the power consumption); however, this aspect is not considered in our evaluation.

The two independent plots in Figure 3 show the behavior for *GP6M2–REGz* and *GP6M3–REGz* on the benchmark set. Notice that adding one memory port reduces both the number of cycles required to execute the loops and the actual execution time. The memory traffic is higher because more ILP is exploited, more spill is required and therefore more pressure is put on the memory ports. However, the differences tend to reduce as the register file size increases (due to the reduction in spill code). For instance, the additional memory port produces a speed–up of 19% in a configuration with 16 registers and 12% in a configuration with 32 registers.

In conclusion, the designer of a VLIW microarchitecture would like to obtain the IPC reported by a configuration with a large number of registers (e.g. *GP6Mx–REG128*) and with the cycle time of configurations with a low number of registers (e.g. *GP6Mx–REG16*). In this paper we present an alternative design for the register file that tries to achieve both things. However, before going into detail, we analyze the influence of the memory latency in performance and register requirements.

3.3. Memory latency

In this subsection we analyze the effect of the memory latency on register pressure. A range of values between 2 and 32 cycles is considered. Although very high values may seem unrealistic for the first level of cache, they are included in our analysis for the following reasons. First, technology trends indicate that the gap between the processor cycle time and the cache hit latency will increase; therefore, considering memory latencies in the range of 4–8 cycles should be possible in the near future. Second, some current designs are proposed so that certain instructions (e.g. floating point) bypass the first level of cache; in this case, observed latencies in the range of 8–16 cycles are reasonable. And third, previous research has shown that scheduling load operations with miss latency (i.e. with the second–level hit latency) may produce better schedules than scheduling them with hit latency [24]. Scheduling load operations with their hit latency generates a valid schedule that forces the processor to stall the execution of a whole long instruction whenever a cache miss occurs on it. Binding prefetching (also known as early prefetching) consists in scheduling load operations assuming their cache miss latency. This ensures that data will always be available when needed. However this early availability increases the lifetime of values brought from memory and, as a consequence, the register pressure.

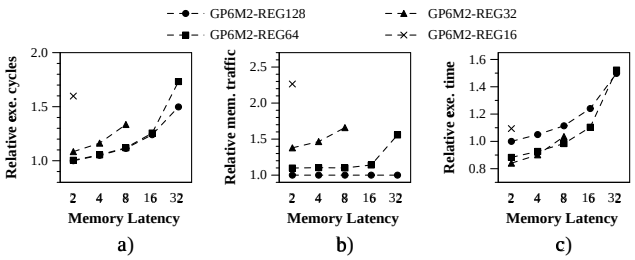

Figure 4. Behavior for different values of memory latency relative to the baseline *GP6M2–REG128*.

Figure 4 shows the behavior for different values of the memory latency. First of all, notice that some of the plots do not have values for all the latency values. When the memory latency increases, our software pipeliner fails to find a valid schedule for some of the loops: the register pressure is so high that the addition of spill code does not effectively reduce it (in fact, the register requirements of the spill instructions can not be absorbed by the registers available). These loops have to be scheduled using a non–pipelined scheduler and for this reason, have not been considered.

Second, notice that the execution cycles increase with the memory latency. This is due to the additional register pressure caused by the load instructions and to the increase in the latency of critical recurrences (i.e. cycles in the dependence graph that limit the *II*) which includes at least one memory operation.

Finally, the trade–off between IPC and cycle time is also noticeable in Figure 4. For small values of the memory latency (e.g. 2 or 4 cycles), the *GP6M2–REG32* register file performs best. This conclusion is applicable to systems in which the processor/memory gap is small and use schedulers that consider hit latency for memory instructions. For latency values in the range 8 to 16 cycles (e.g. future systems with a high processor/memory gap or when using schedulers that apply binding prefetching), configuration *GP6M2–REG64* is able to better trade–off IPC and cycle time. Configuration *GP6M2–REG128* become interesting when we consider very high values of the memory latency (i.e. when considering binding prefetching and future technologies that lead to a high processor/memory gap).

4. New register file organization

In the previous section we concluded that a register file with a high number of registers and a large number of access ports is required in order to effectively exploit ILP. However, the cycle time obtained from the access time to the register file easily offsets the gains obtained in terms of instructions executed per cycle.

In order to have a register file organization with a large number of ports and a low access time, this paper proposes a two-level hierarchical register file organization, as shown in Figure 5.a. The first level, named *R1*, has a small capacity but a high number of ports (in order to feed all the functional units); this will permit a design with a small access time. The second level, named *R2*, has a higher capacity in order to avoid the degradation that might be caused by the small capacity of *R1*, to reduce the necessity of spill and to improve the performance through the use of binding prefetching. This level interacts with *R1* and with the first level of cache memory; *R2* is not directly accessible by the functional units so it has a small number of ports. This level will be designed so as to have as many registers as possible without penalizing the access time determined by *R1*.

The main drawback of the two–level register file organization is the increase in the latency observed between memory and functional units and the increase in the number of instructions in the program. In this organization, a memory *load* operation brings data to *R2*, so that an extra number of cycles is required in order to move data from *R2* to *R1*. The same happens in a *store* operation which requires a first move from *R1* to *R2* followed by the memory access. This extra latency increases the *II* when a loop has a critical recurrence including at least one memory operation.

In a VLIW processor, this movement between *R1* and *R2* is controlled by the compiler. Two new operations are needed to move data between register file levels: *loadR* and *storeR*. The compiler inserts a *loadR* after the original *load* operation. Similarly, the compiler inserts a *storeR* operation before the original *store* operation. The compiler can also move data between levels in order to spill values when the pressure in *R1* is higher than its capacity.

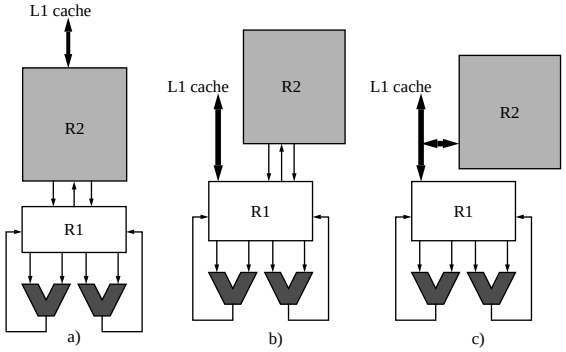

Figure 5. a) Two–level hierarchical register file; b) and c) two alternative CCM organizations.

The explicit management of these data movements requires modifications in the algorithm used for register allocation and spilling. The proposed algorithm first allocates registers in *R1* using the wands-only strategy with end-fit and adjacency ordering [21]. If there are not enough registers in *R1*, spill code between *R1* and *R2* is inserted. Spill code is added using the techniques proposed in [28] and using only the *loadR* and *storeR* instructions. Once the register allocation and spill is completed for *R1*, the algorithm proceeds with *R2*. In *R2* the algorithm has to fit the registers required by data brought from the memory as well as the registers required by the spill of *R1*. Register allocation and spilling is performed using the same techniques as for *R1*. Notice that this spill code temporarily stores values in memory; however, the high capacity of our register file organization reduces the necessity for this.

4.1. Design issues

In a two-level hierarchical register file there are three parameters that influence the final performance (without varying other parameters such as the number of functional units and memory ports). These parameters are: a) number of registers in *R1*, b) number of read and write ports between *R1* and *R2* and c) number of registers in *R2*. We assume that the number of registers in each level is a power of two.

In order to find the most appropriate value for *R1*, 3 possible configurations have been explored (8, 16, and 32):

- For the 8 registers configuration, there are so few registers that our software pipeliner fails to schedule many of the benchmark loops due to the high number of functional units available in the architecture evaluated; although possible schedules could be found, they would have very high values for the *II*.

- For the 16 registers configuration, the software pipeliner is able to schedule all the loops in the benchmark set. This therefore is the minimum size that should be considered.

- For the 32 registers configuration, the software pipeliner is also able to schedule all the loops. However, its access time is slightly higher than the access time of a monolithic 32 register file. Moreover, even assuming an infinite number of ports and an unlimited number of registers in the second level, the performance (in number of cycles) is worse than the performance of the monolithic design with 32 registers.

Therefore, 16 registers have been included in *R1* because this allows the schedule of all the loops and has a competitive cycle time compared to the design with 32 registers which achieved the best execution time with small memory latencies (Figures 3.c).

In order to determine the number of read and write ports between *R1* and *R2*, we have first assumed that *R1* has 16 registers, the number of ports between *R1* and *R2* is unlimited and that *R2* has also an unlimited number of registers. Figure 6 shows the cumulative distribution of loops that require, on average, a specific number (e.g. a loop that requires 4 read ports in cycle 0, 2 in cycle 1 and 0 in cycle 2 requires, on average, 2 ports per cycle). For instance, 80% of the loops require an average of 2 load ports at most. Since software pipelining tries to saturate all the resources available, it is sufficient to have as many ports as the maximum average required to schedule all the loops. In Figure 6, it should be noted that the scheduler never uses more than 2 store ports and 4 load ports.

The parameters decided at this point, together with the number of functional units, determine the access time to *R1*. The upper part of Table 2 estimates this. The capacity of *R2* is selected as the maximum capacity with an access time smaller than (or equal to) the access time of *R1*. This means that the number of cycles to execute *loadR* and *storeR* operations is one cycle. Notice that we could use more registers in *R2* and schedule these move operations with the appropriate latency. The lower half of Table 2 shows an estimate of the access time for *R2* with 64 and 128 registers. From these figures, 64 registers in *R2* fulfills the requirements (although using 128 registers would only increase the cycle time by 2% but would require more area and power). Moreover, we have experimentally proven that with 64 registers, the spill to memory is negligible.

In summary, the two-level hierarchical register file organization proposed in this paper (named *TWO16*) has the following parameters: 16 registers in the first level *R1*; 64 registers in the second level *R2*; 2 store ports from *R1* to *R2*; and 4 load ports from *R2* to *R1*.

4.2. Register file area and power dissipation

In this subsection we estimate the area and power dissipation for the register file configuration proposed in this section and compare it with that of a monolithic organization. The model described in [23] is used to compute these figures for a monolithic register file (see Table 1 for configurations *REG16*, *REG32*, *REG64*, and *REG128*). For the two–level organization, we assume that they are computed as the sum of the individual figures for *R1* and *R2*.

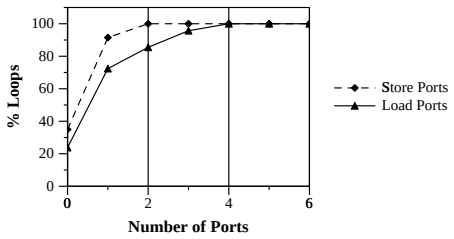

Figure 6. Analysis of requirements: number of read and write ports between *R1* and *R2*.

Table 2 shows the area estimated for configuration *TWO16*. Notice that in the monolithic organization, the area of *REGz* is linear with *z* because all of them have the same number of ports. The area of *R1* in configuration *TWO16* is the same as *REG16* with 3 memory ports; the area of *R2* is smaller than the area of *REG64* because it has less access ports. From these area estimations, we conclude that *TWO16* is 14% larger than *REG32* and 43% smaller than *REG64* in processor configuration *GP6M2*.

Regarding power dissipation, the model in [23] gives the dissipation per access to the register file. The peak power dissipation is computed multiplying this value by the maximum number of accesses per cycle that the register file supports. From these estimations, we conclude that the power dissipation of *TWO16* is 4% greater than *REG32* and 40% less than *REG64* in processor configuration *GP6M2*. A complete evaluation of power consumption is necessary in order to perform an accurate comparison of both organizations. In this evaluation, the power consumption of other parts of the system should be considered, such as the instruction and data caches. For instance, notice that the traffic with the data cache is reduced (due to the reduction of spill operations) and therefore its power consumption is, too. However, the number of accesses (and their width) to the instruction cache is increased (due to the the higher values of the *II* and the extra length of the instruction). This evaluation is part of our current work.

5. Related register file organizations

The organization and management of the register file has been a subject of research in the past. The main idea behind all this research is to strike a compromise between the parameters related to storage capacity, area, access time and power dissipation of the register file. As we have concluded in Section 3, the single monolithic register file which has been traditionally used in microprocessors does not scale well when the register requirements are high, i.e. when using either aggressive processor configurations or instruction scheduling algorithms such as software pipelining. Recently, [23] developed a taxonomy of register architectures and evaluated it for media and signal processors with a large number of arithmetical units.

Size	Ports	**Cycle time**	**Area**	**Power**
R1	R/W	ηs (relat.)	$\lambda^2 * 10^6$ (relat.)	W (relat.)
16	14/10	1.1489 (0.695)	0.774 (0.145)	2.818 (0.270)
R2	R/W	ηs (relat.)	$\lambda^2 * 10^6$ (relat.)	W (relat.)
64	6/4	1.0205 (0.618)	0.745 (0.140)	0.840 (0.081)
128	6/4	1.1735 (0.711)	1.491 (0.280)	1.648 (0.158)
TWO16		1.1489 (0.695)	1.519 (0.285)	3.658 (0.351)

Table 2. Access time, area and power for *R1* and *R2* in the two–level register file organization (absolute and relative to the baseline configuration *GP6M2–REG128*.

5.1. One–level (distributed/partitioned) register file organization

Some proposals for alternative register file organizations focus on the use of multiple register banks so as to reduce the number of ports needed by each register bank. Register banks can be used to replicate the available registers: fully replicated as in some current out–of–order microprocessors [13, 27] or partially replicated for VLIW processors [15]. Register banks can also be used to distribute the total number of registers [3, 10]. The partition limits the connectivity between register banks and functional units thus creating a set of clusters; different topologies can be used to connect clusters (global shared buses [3] or rings [10]).

Other proposals do not restrict the connectivity between the register banks and the functional units. Banks may be designed with different characteristics or purposes. For instance, [19] proposed to reduce the number of registers required in the main register bank by adding a second port–limited bank (called the sack) with only one read port and one write port. Each bank tries to capture values with different locality properties.

5.2. Hierarchical register file organization

The hierarchical organization of the register file is not new. In these organizations not all banks can directly feed the functional units and/or memory. The level close to the functional units can be designed with less capacity and therefore, small access time. For instance, the CRAY–1 [25] included, for the scalar functional units, two independent banks (one for addresses and another for data) each with a two–level organization: the lower level (close to the functional units) had 8 registers and the upper level had 64 registers able to do block transfers with the memory. The upper level had no connection with the functional units; both register files were connected to the main memory. The upper level was used by the compiler to optimize data locality.

In a different context, [26] proposed the hierarchical design of a register file. Their register file had 1024 registers in three levels: ”close” level with 5 registers, ”middle” level with 59 registers and ”distant” level with 960 registers. In this hierarchy, the latency to access a given register level is less when closer to the functional units.

In parallel with the work presented in this paper, a caching mechanism for the register file in dynamically scheduled processors has been proposed [7]. The organization is hierarchical and the allocation of values to registers is performed at runtime. The mechanism is transparent to the compiler and acts as a cache for the register file.

5.3. Local memory for spill code

A register file organization oriented towards capturing spill code in DSP processors is proposed in [6]. The small (and fast) local memory, which is available in some of these processors, is used as a holding place for spilled values. They called it CCM (compiler–controlled memory). The CCM does not share the address space with the cache memory. Spilling to the CCM removes spill traffic from the path to main memory and eliminates the cache pollution introduced by spill operations. Their proposal was evaluated in a simple machine model that issued one instruction per cycle. In order to do a more realistic comparison with our proposal, Section 6.1 evaluates their performance with our machine configurations, our scheduler and benchmarks.

Figure 5 shows two possible implementations for the CCM proposal. Figure 5.b shows an implementation in which the buses connecting *R1* to *R2* and memory are independent. In this case, memory and spill operations can be scheduled in the same cycle. However, the access time to the register file is increased due to the additional ports. For this implementation, two configurations are evaluated with 16 registers in *R1*: *CCMind(16–6)* with 4 write and 2 read ports and *CCMind(16–3)* with 2 write and 1 read ports.

Figure 5.c shows an alternative implementation in which the *R2* and memory are accessed through the same bus (shared bus). In this case the number of ports is not increased and therefore the cycle time is not affected. However, the modulo scheduler has more constraints in order to schedule memory and spill operations because they require the same resource. For this implementation, two configurations are evaluated: *CCMsh(16)* and *CCMsh(32)* with 16 and 32 registers in *R1*, respectively.

Table 3 shows the cycle time estimated for the 4 different configurations assumed for the CCM. We have considered that the *R2* does not limit the cycle time and is large enough to ensure that all the spill traffic is kept in *R2*.

6. Performance evaluation

This section evaluates the performance of the hierarchical register file organization *TWO16* and compares it with two monolithic designs *REG32* and *REG64*. First, the evaluation is carried out assuming an ideal memory system (i.e. always hit in the cache); the performance of *TWO16* is also compared with the performance for two possible implementations of the CCM organization [6]. Finally, the evaluation is carried out assuming a real memory system.

	Size	Ports		**Cycle time**
Configuration	R1	Read	Write	ηs (relat.)
CCMind(16–6)	16	20	14	1.1844 (0.717)
CCMind(16–3)	16	19	12	1.1580 (0.701)
CCMsh(32)	32	18	10	1.2805 (0.775)
CCMsh(16)	16	18	10	1.1304 (0.685)

Table 3. Access time for a set of CCM register file implementations.

6.1. Ideal memory system

Figure 7.a shows the execution cycles spent in the loops for the three possible configurations. Notice that *TWO16* always requires more cycles to execute the benchmark set than *REG64*. Compared to *REG32*, *TWO16* requires less execution cycles when the latency to memory is higher than 4 cycles. This is due to the extra cycles needed to move data to/from memory and the small capacity of the first level *R1*. Figure 7.b shows that *TWO16* practically absorbs all the memory traffic incurred by spill operations.

Figure 7.c shows the relative execution time when the cycle time is factored in. Notice that *TWO16* outperforms both *REG32* and *REG64* configurations for all memory latencies. For low latencies (2 cycles) *TWO16* is 6% (11%) faster than the *REG32* (*REG64*) configuration. However if the memory latency is larger the speed–up increases. For instance, for 8 cycles of latency, *TWO16* shows a speed–up of 18% compared to *REG32*; for 32 cycles of latency the speed–up is 22% compared to *REG64*.

The low cycle time achieved by the hierarchical configuration makes it very competitive even assuming systems with the monolithic register file and more ports with the memory system. Figure 7.d–f shows the results when configuration *GP6M3* is considered. Notice that, in terms of relative execution time, configuration *REG32* is better only if the memory latency is small (2 cycles); for higher latencies, *TWO16* performs slightly better (4%) than *REG64* with less ports in the memory hierarchy. Moreover, notice that the scheduler always finds a possible software pipelined schedule for all the values of memory latency (it is not able to do so for *REG32* when the memory latency is higher than 8 and for *REG64* when the latency is higher than 16.

For the four configurations of the CCM proposal previously defined, we have repeated the simulations and their results are shown in Figure 8. The plot for memory traffic has

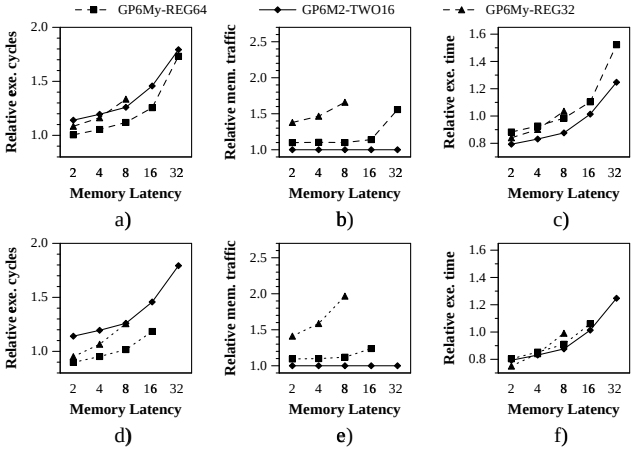

Figure 7. Behavior of *GP6M2–REG32/64* (first row), *GP6M3–REG32/64* (second row) and *GP6M2–TWO16*, all of them relative to *GP6M2–REG128*.

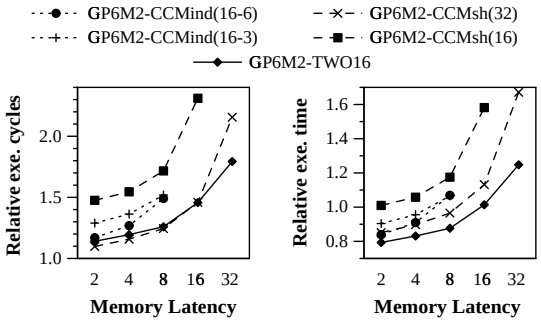

Figure 8. Behavior of some alternative implementations of CCM compared to *TWO16*.

been eliminated because it always remains constant (i.e. all configurations completely eliminate spill to memory). Notice that, in terms of number of execution cycles, the register file organization proposed in this paper performs better than the two *CCMind* configurations evaluated, much better than *CCMsh* with 16 registers in *R1* and similar to *CCMsh* with 32 registers in *R1* (up to 16 cycles). In terms of execution time, the performance achieved by the proposal in this paper is always higher and less sensitive to cache miss latency.

6.2. Real memory system and binding prefetching

In this section we analyze the behavior of the proposed register file organization in a real memory environment. As mentioned in Section 2, hit latency for load operations is 2 cycles and miss latency may have three possible values: low (10 ηs), medium (20 ηs) and high (40 ηs). Cache–miss latencies in cycles are computed assuming the corresponding cycle time for each configuration of the register file. For example, Table 4 shows the cache miss latency in cycles for the *GP6M2* configuration.

The evaluation breaks down the total number of cycles and execution time into two components: useful (i.e. when the processor is doing useful work) and stall (i.e. when the processor is blocked waiting for a cache miss to complete the access). All performance figures in this section are relative to the number of useful cycles of configuration *GP6M2–REG32* with small cache–miss latency.

The modulo scheduler used in our experimental framework can assume either hit latency to schedule memory load operations or apply binding prefetching. Scheduling with hit latency minimizes the register pressure and theoretically increases performance. This generates a valid schedule that stalls the processor whenever a cache miss occurs or when-

	Cycle time	Cache–miss latency		
Configuration	ηs	Small	Medium	High
GP6M2–REG32	1.2805	8	16	32
GP6M2–REG64	1.4513	7	14	28
GP6M2–TWO16	1.1489	9	18	35

Table 4. Cycle time and cache–miss latencies for 32, 64 and two–level register files.

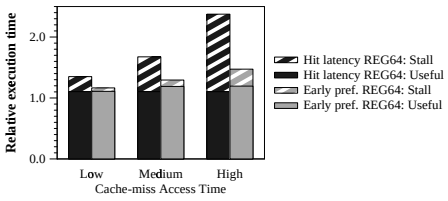

Figure 9. Execution time when scheduling loops using hit latency or selective binding prefetching.

ever a dependent instruction needs the datum brought from memory (in case of lockup–free caches). Binding prefetching can be used to tolerate the latency of these cache misses [2]. Binding prefetching consists in scheduling the *load* instructions assuming cache miss latency. Binding prefetching does not increase memory traffic but increases register pressure, as shown in Section 3.3. However, the higher capacity of the proposed *TWO16* organization allows us to apply aggressive prefetching techniques.

In this paper we use a selective binding prefetching approach. The algorithm assumes that those load operations included in recurrences as well as spill load operations are scheduled assuming hit latency. All other load operations are scheduled assuming miss latency. Those loops which execute a small number of iterations are also scheduled assuming hit latency for all their memory load operations (in order to avoid long prologues and epilogues in the software pipelined code). For instance, Figure 9 compares the performance of configuration *GP6M2-REG64* when loops in our benchmark set are scheduled either assuming hit latency or when applying selective binding prefetching. Notice that binding prefetching generates schedules that noticeably reduce the execution time (for instance, up to 40%). With binding prefetching, the number of useful cycles is increased and the number of stall cycles reduced; notice that stall cycles are not completely eliminated due to the misses that may happen in recurrences and in short loops.

Assuming that binding prefetching always results in better schedules, we proceed with a comparison of the monolithic and two–level register organizations. Figure 10.a shows the execution cycles relative to the useful cycles of the *REG32* configuration. Notice that *GP6M2–REG64* requires less cycles to execute than its *TWO16* counterpart. However, the influence of the cycle time offsets this result in favor of the two–level organization, as shown in Figure 10.b. For instance, *TWO16* reduces the execution time of *REG64* in a range of 10–14%. Compared to *REG32*, the reduction in execution time is as much as 38%.

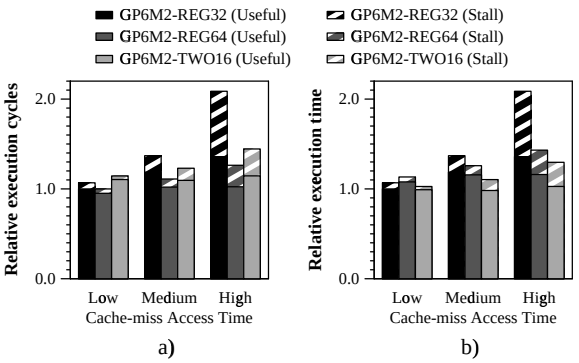

Figure 10. Performance evaluation for configuration *GP6M2* with three different configurations for the register file: *REG32*, *REG64* and *TWO16*.

7 Conclusions

High-performance microprocessors are currently designed to exploit the inherent ILP available in most applications. The techniques used in their design and the aggressive scheduling techniques tend to increase the register requirements of the loops. If more registers than those available in the architecture are required, some actions are required to reduce this pressure, at the expense of a performance degradation.

The monolithic register file design that has traditionally been used to interconnect functional units and provide short–term storage does not scale well when the architecture becomes aggressive, which is the current trend in the design of high–performance microprocessors. Cycle time, area and power consumption are the factors that limit its performance and usability. These limit the size of the register file and therefore introduce some degradation because of the addition of spill code. This degradation could be avoided if high–capacity register file organizations could be included in VLIW designs, but without having a negative impact on the cycle time, area and power of the processor.

In this paper we have proposed a two–level hierarchical register file organization that combines high capacity and low access time. For the configurations and workload evaluated, the best register file configuration consists of 16 registers in the first level and 64 registers in the second level. The high capacity of the register organization reduces the amount of register spilling and therefore its additional memory traffic. It also allows the use of more aggressive prefetching techniques to hide the negative effect of high miss latencies. The proposed organization achieves a cycle time 2% greater than a monolithic register file configuration with 16 registers. For one of the processor configurations and workload evaluated in this paper, this new organization reduces the execution time by 10–14% when compared to a monolithic organization with 64 registers; this speed–up is obtained with a 43% reduction in area. Compared to a monolithic organization with 32 registers, the reduction in execution time is up to 38% with a 14% increase in area.

In addition, the new organization requires much less peak power (40%) than a monolithic organization with 64 registers and slightly more (4%) than 32 registers. We estimate that this small difference in peak power will result in

less energy consumption because of less memory accesses caused by spill code. However, the additional instruction accesses may reduce this advantage. Further research is required to accurately evaluate energy consumption.

In some way, the hierarchical organization proposal in this paper could be defined as a functional clustering in which one cluster is composed of the functional units and a small register file with a large number of ports. The second cluster is composed of the memory units and a higher register file with the less ports. We are currently extending this idea in order to have multiple clusters for the functional units with smaller register files.

References

[1] 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.

[2] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40–52, April 1991.

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

[4] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc., 18th Internat. Symp. on Computer Architecture*, pages 266–275, 1991.

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

[6] K. D. Cooper and T. Harvey. Compiler–controlled memory. In *Proc., Eighth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 100–104, October 1998.

[7] J. Cruz, A. Gonzalez, M. Valero, and N. Topham. Multiple–banked register file architectures. In *Proc., 27th Annual Internat. Symp. on Computer Architecture*, June 2000.

[8] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.

[9] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 338–349, November 1995.

[10] M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered vliw architectures. In *Proc., 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'1998)*, pages 386–391, March 1998.

[11] R. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the 6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.

[12] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1/2):229–248, 1993.

[13] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

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

[15] J. Llosa, M. Valero, and E. Ayguadé. Non-consistent dual register files to reduce register pressure. In *1st Symposium on High Performance Computer Architecture*, pages 22–31, January 1995.

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

[17] J. Llosa, M. Valero, and E. Ayguadé. Quantitative evaluation of register pressure on software pipelined loops. *International Journal of Parallel Programming*, 26(2):121–142, April 1998.

[18] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 350–360, November 1995.

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

[20] B. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7(1/2):9–50, July 1993.

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

[22] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[23] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proc., 6th High-Performance Computer Architecture (HPCA–6)*, pages 375–386, January 2000.

[24] J. Sanchez and A. Gonzalez. Cache sensitive modulo scheduling. In *Procs. of the 30th Annual Int. Symp. on Microarchitecture (MICRO-30)*, pages 338–348, December 1997.

[25] D. Siewiorek, C. Bell, and A. Newell. *Computer Structures: Principles and Examples.* MacGraw-Hill, Pittsbutgh, Pennsylvania., 1982.

[26] J. Swensen and Y. Patt. Hierarchical registers for scientific computers. In *International Conference on Supercomputing*, pages 346–353, July 1988.

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

[28] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved spill code generation for software pipelined loops. In *Procs. of the Programming Languages Design and Implementation (PLDI'00)*, June 2000.