Impact on Performance of Fused Multiply-Add Units in Aggressive VLIW Architectures

David López, Josep Llosa, Eduard Ayguadé and Mateo Valero
Department of Computer Architecture. Polytechnic University of Catalunya.
UPC-Campus Nord, Mòdul D6. Jordi Girona 1-3, 08034 Barcelona (Spain)
E-mail: { david | josepll | eduard | mateo }@ac.upc.es

Abstract

Loops are the main time consuming part of programs based on floating point computations. The performance of the loops is limited either by recurrences in the computation or by the resources offered by the architecture. Several general-purpose superscalar microprocessors have been implemented with multiply-add fused floating-point units, that reduces the latency of the combined operation and the number of resources used. This paper analyses the influence of these two factors in the instruction-level parallelism exploitable from loops executed on a broad set of future aggressive processor configurations. The estimation of implementation costs (area and cycle time) enables a fair comparison of these configurations in terms of final performance and implementation feasibility. The paper performs a technological projection for the next years in order to foresee the possible implementable alternatives. From this study we conclude that multiply-add fused units may have a deep impact in raising the performance of future processor architectures with a reasonable increase in cost.

1. Introduction

Current high-performance microprocessors rely on hardware and software techniques to exploit the inherent instruction-level parallelism (ILP) of the applications. These processors make use of deep pipelines in order to reduce the cycle time and wide instruction issue units to allow the simultaneous execution of several instructions per cycle.

Very Long Instruction Word (VLIW) architectures are oriented to exploit ILP. In a VLIW architecture, an instruction is composed of a number of operations that are issued simultaneously to the functional units (i.e. the scheduling is performed at compile time so the dispatch phase is very simple). Although there are few commercial general purpose VLIW processors, these architectures have been widely used in the DSP arena (as in the Texas Instruments ‘C6701 [21] and Equator Map1000 [22]), they have been subject of research in the last years and will constitute the core of future designs [19].

The static nature of VLIW schedulings require good compilation techniques to effectively exploit the ILP available in real programs [5]. Software pipelining [10] is a compilation technique that extracts ILP for the innermost loops by overlapping the execution of several consecutive iterations. In a software pipelined loop, the number of cycles between the initiation of successive iterations (termed *Initiation Interval*) is bounded either by the recurrences in the dependence graph or by the resource constrains of the target architecture [4][23][24].

Several techniques have been proposed to increase the performance of loops bounded by the resource constrains. The loops performance can be augmented by increasing the number of functional units (replication technique), by exploiting data parallelism at the functional unit level (like in vector processors [27] or, in superscalar and VLIW processors, using the widening technique [13][14][20]), or by using functional units that can perform multiple operations as a monolithic operation (e.g. *fused multiply and add* FMA floating-point units perform a multiplication and a dependent addition as a single operation).

As the number of transistors on a single chip continues to grow, more hardware can be accommodated on a chip, so future designs will use these techniques to exploit ILP aggressively. Unfortunately, most of these techniques focus on increasing the performance of resource-bound loops. Consequently, the more aggressive become the architectures, more critical become the recurrences.

The FMA operation reduces the latency of the recurrences. Several current microprocessors implement this operation (like the IBM RS/6000 [7] and POWER2 [28], and the MIPS R8000 [6] and R10000 [30]). This paper studies the influence of fused multiply-add functional units (fusion technique) in future ILP aggressive architectures. We study the maximum ILP achievable using this technique, combined with other techniques that increase the performance of resource-bound loops. In order to perform a fair comparison of the different techniques, it is mandatory to evaluate the effect in the final performance of several factors: the influence of the compiler, the influence of the register file size, and the hardware cost in terms of chip area and cycle time.

The area cost defines those configurations that could be implemented in the next microprocessor generations, according to the predictions of the *Semiconductor Industry Association* [26]. For each generation we estimate the performance of a set of implementable configurations taking into account the number of cycles required to execute the programs and the cycle time. From this study we conclude that the fusion technique has a significant effect on the final performance of aggressive configurations. The technique has a good theoretical performance, but also reduces the register pressure, it gives more opportunities to the compiler to find an optimal scheduling, and also has good performance/cost efficiency.

All the evaluations have been performed for VLIW architectures and numerical programs. Our workbench is composed of 1180 loops that account for 78% of the execution time of the Perfect Club [2]. The loops have been obtained using the experimental tool Ictíneo [1] and software pipelined using *Hypernode Reduction Modulo Scheduling*

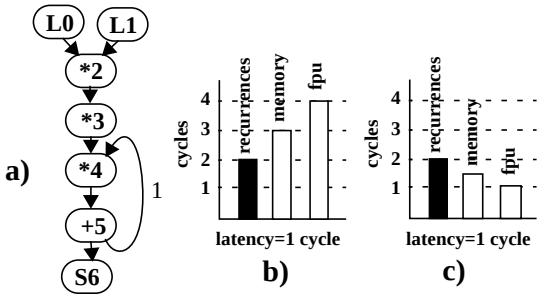

Figure 1: a) Sample loop dependence graph and limits for two processor configurations: b) 1 bus and 1 functional unit, c) 2 buses and 4 functional units.

[17][18], a register pressure sensitive heuristic that achieves near optimal schedules. Register allocation has been performed using the wands-only strategy and the end-fit with adjacency ordering [25]. When a loop requires more than the available number of registers, spill code is added and the loop is rescheduled [15][16].

The organization of the paper is as follows: Section 2 describes the replication, widening and fusion techniques and illustrates, with an example, their performance limits. Section 3 describes the FPUs that implement the FMA operation and outlines its cost. Section 4 evaluates the performance of a set of processor configurations where the studied techniques are combined. First we perform an estimation of the theoretical ILP limits of the techniques, assuming perfect scheduling, a register file of infinite size and a perfect memory. Then we evaluate the effect of the scheduler and having a finite register file. Finally, we take the costs into account and study the performance/cost trade-offs of the techniques. Section 5 summarizes the main conclusions of this work.

2. Motivating example

In a software pipelined loop, the number of cycles between the initiation of successive iterations is named *Initiation Interval* (*II*). The *II* defines the maximum performance that can be obtained from the loop and is bounded either by resources constrains in the architecture (*ResMII*) or by cyclic dependence chains (recurrences) in the dependence graph (*RecMII*). Its lower bound is termed the *Minimum Initiation Interval* (*MII*) and is computed as *MII = max (ResMII, RecMII)*.

In this section we use a sample loop (whose dependence graph is shown in Figure 1.a.) to show these bounds and possible alternatives to reduce them. In the Figure, nodes represent operations (memory accesses or arithmetic computations) and edges represent data dependences between pairs of nodes. There are 3 memory operations (loads L0 and L1, and store S6) and 4 arithmetic operations (products *2, *3, *4 and addition +5). All dependences are intra-loop dependences (i.e. occur between two operations performed in the same iteration and have distance 0) except for dependence (+5, *4) in which the result of +5 is used by *4 one iteration later (loop-carried dependence of distance 1). In this graph, there is a recurrence composed of edges (*4, +5) and (+5, *4) spawning over one iteration.

Figure 1.b illustrates the factors that contribute to *ResMII* and *RecMII*, assuming an architecture with a single memory unit and a single arithmetical unit (all of them fully pipelined and with a latency of one cycle):

• Resources. In this case, 3 cycles are required at least to execute the 3 memory operations in the memory access unit (bar labelled *memory*) and 4 cycles are required at least to execute the 4 arithmetical operations in the arithmetical unit (bar labelled *fpu*). Therefore, the arithmetic operations put higher constraints in the execution of the loop and limit the *ResMII* to 4.

• Recurrences. There is a single recurrence in the loop that may limit its execution. For the architecture being considered, 2 cycles are required to sequentially execute the operations within the recurrence, and therefore the *RecMII* is 2 (bar labelled *recurrences*).

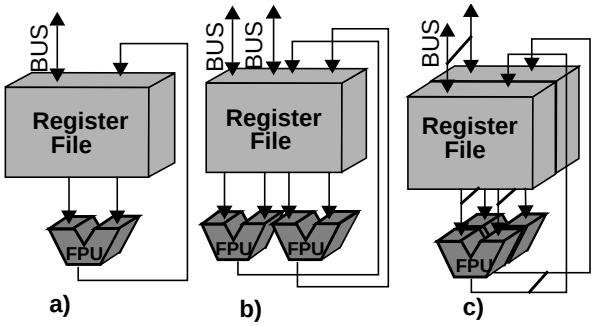

Figure 2: Different processor configurations based on replication and widening: a) base configuration, b) replication and c) widening.

For a given architecture, and depending on the values of *ResMII* and *RecMII*, loops can be classified into three groups: *balanced* (*ResMII = RecMII*), *resource-bound* (*ResMII > RecMII*) and *recurrence-bound* (*ResMII < RecMII*) loops. For example, our working example fits in the resource-bound cathegory limited by the arithmetic operations. Making the architecture more aggressive in terms of number of functional units can convert resource-bound loops into balanced or recurrence-bound loops. Also, reducing their latency can convert recurrence-bound loops into balanced or resource-bound loops. In the next subsections we analyse different architectural alternatives to improve each of these 2 factors.

2.1 Resource-bound loops

In order to increase the performance of resource-bound loops, the resources of the processor must be increased. In this section we analyse two alternatives to make the architecture more aggressive: resource *replication* and *widening*. On one side, resource replication consists on increasing the number of functional units available in the processor. On the other side, resource widening [13] consists on increasing the number of operations that each functional unit can simultaneously perform per cycle (i.e. functional units that operate with short vectors).

For example, Figure. 2.a shows a base configuration in which we have one bus and one floating-point functional unit (FPU). In this case, one memory and one arithmetical operation can be issued per cycle. Higher performance can be obtained by adding another bus and another FPU (replication technique, Figure 2.b); in this case, two independent memory accesses and two independent arithmetic operations can be issued per cycle. The same peak performance can also be obtained by duplicating the width of both, the bus and the FPU (widening technique, Figure 2.c); in this case, two memory and two arithmetical operations can also be issued per cycle. However, widening

is less versatile than replication, because it requires the operations to be *compactable* (i.e. the same arithmetic operation has to be performed in consecutive iterations without data dependences between them, or access to consecutive memory locations in the case of memory operations). This analysis of compactability can be done at compile time [12][14]. Although replication enables the exploitation of more ILP than widening, its larger costs (in terms of area and cycle time) precludes the use of high degrees of replication in favour of a combination of small degrees of replication and widening. A detailed performance/cost analysis of different future processor configurations based on a combination of replication and widening can be found elsewhere [13].

The use of aggressive configurations (using replication and/or widening) for the processor core does not affect the performance of recurrence-bound loops and may covert balanced or compute-bound loops into recurrence-bound loops. In our working example, when the number of resources is increased (as shown in Figure 1.c), the bars due to memory and arithmetic operations are reduced. Notice that at this point the dominant bar is the one due to the recurrences, the performance will not improve even in an architecture with an unbounded number of resources.

Figure 3 shows, for our workbench, the percentage of time spent in recurrence, compute and memory bound loops for different processor configurations (executed on an HP 9000/735 workstation and compiled with the +O3 flag, which performs software pipelining among other optimizations). Each configuration *xZ* includes Z functional units and Z buses, with a latency of 4 cycles. For a configuration with 1 FPU and 1 bus (as the one shown in Figure 2.a) 61.8% of the total time is spent in compute-bound loops, 30.7% in memory-bound loops and only 7.5% in recurrence-bound loops. This indicates that the loops of our workbench are slightly compute-bound. Notice that the percentage of time spent in loops that become recurrence bound increases when the aggressiveness of the architecture increases. For configurations greater than x4, more than 50% of the total time is spent in recurrence-bound loops. As a consequence, techniques to improve the performance of this kind of loops are necessary in order to impact the final performance of aggressive architectures.

2.2 Recurrence-bound loops

In order to improve the performance of loops bounded by recurrences, the number of cycles needed to perform the operations in the recurrence has to be reduced. This reduction can be achieved either by reducing the latency of the functional units or by solving complex operations in the same amount of time. The later option has been included in the design of some current microprocessors with functional units that execute a multiply instruction and an associated

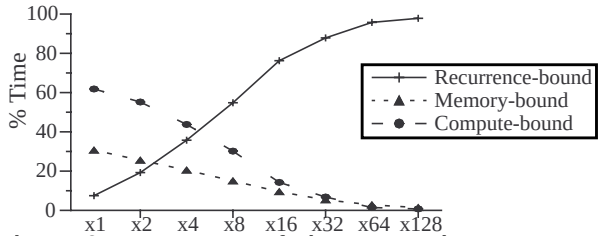

Figure 3: Percentage of time spent in recurrence, compute and memory bound loops for different processor configurations.

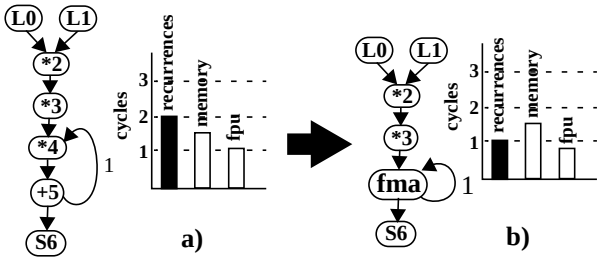

Figure 4: Graph transformation and impact in performance limits in an architecture with 2 buses and 4 FPU: a) without FMA and b) with FMA.

addition operations (FMA) as a single instruction. These functional units reduces both the number of cycles needed to execute the recurrence and the number of slots used to schedule these operations.

For example, the dependence graph in our working example has a recurrence composed of two nodes where a product is followed by an addition. Figure 4 shows how the graph is transformed when these two operations are fused in a single FMA operation, thus reducing the number of operations of the recurrence from 2 to 1, and reducing the recurrence limit to 1 cycle.

A static analysis of our workbench reveals that 591 out of the 1180 loops can make use of FMA functional units. For these loops, Figure 5 shows the percentage of time spent in loops that would benefit from having FMA operations for different *xZ* processor configurations. For each one:

- dotted line: it shows the percentage of time spent in compute-bound loops that would benefit from having FMA operations. Notice that in our base configuration, more than 60% of the time is spent in this kind of loops. As configurations become more aggressive, the number of compute-bound loops are reduced, becoming less than 1% in the largest configuration considered.
- dashed line: it shows the percentage of time spent in recurrence-bound loops that would benefit from having FMA operations (i.e. those in which this operation occurs in the most limiting recurrence). As configurations become more aggressive, the percentage of time spent in these loops increase.

The solid line summarizes both effects and represents the total percentage of time spent in loops that would benefit from having FMA operations. Notice that this kind of complex operations are dominant in numerical applications and may affect the performance when aggressive architectures are considered. In order to validate this thesis, this paper analyses the impact of FMA operations in the performance of future aggressive processor configurations based on a performance/cost evaluation where ILP, area and cycle time are taken into consideration.

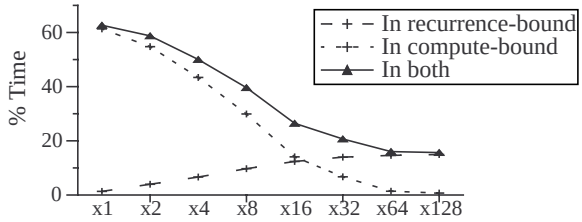

Figure 5: Percentage of time spent in loops than would benefit from implementing the FMA operation for different processor configurations.

3. Cost of Fused Multiply-and-Add FPUs

Some current microprocessors implement FPUs that execute the *fused multiply-and-add* operation (FMA) as a single instruction as T = (A x C) + B. In these FPUs, the floating-point hardware is designed to accept up to three operands for executing FMA instructions, while other floating-point instructions requiring fewer than three operands may utilize the same hardware by forcing constants into the unused operands. In general, FPUs with FMA implementations use a multiply array to compute the AC product, followed by an adder to compute AC+B.

This operation has been implemented in the floating-point units of the IBM RS/6000 [7], IBM POWER2 [28], MIPS R8000 [6] and MIPS R10000 [30] microprocessors. In the MIPS R10000, the FMA operation has been implemented by chaining the multiplier FPU output with the adder FPU input requiring rounding and alignment between them. Therefore the MIPS R10000 requires 2 cycles to compute an add or a mul, and 4 cycles to compute a FMA operation providing no latency benefit. The only benefit is the reduction in instruction bandwidth and in the registers requirements (no register is required to store the intermediate result). On the contrary, processors like the IBM POWER2 implement the FMA operation integrating the multiplier and the adder without rounding and alignment in the middle. Therefore in the POWER2, the FMA operation has the same latency (two cycles) as the individual add or mul operations

Implementing the FMA functional unit in a microprocessor incurs several costs in terms of area and cycle time. With respect to the FPUs, the area required for the extra hardware needed to implement the FMA operation is practically negligible because the area of a general-purpose floating-point unit is mainly governed by the area of the multiplier [9]. The main additional cost is due to the associated register file. The overall size of a register file is determined mainly by the size of a register cell. The other components that are needed to access the register file typically represent less than 5% of the area required by the register cells [11].

The area of the register cell grows approximately as the square of the number of ports added because each port forces the cell to increase both the height and the width [8][11][12]. Implementing the FMA operation requires one additional read port for every FPU, so it results in an increase of the register file size. To illustrate this cost, Table 1 shows the area cost of one configuration (named A) with 2 buses and 2 FPUs. In this case, each bus requires 1 read and 1 write ports, and each FPU requires 2 read and 1 write ports. Assuming a register file of 32 registers, the register file cost results in $32.08 \times 10^6\ \lambda^2$ (the details of this cost model can be found elsewhere [12]). If the FPUs implement the FMA operation, one more read port is required for each FPU (configuration B), resulting in a register file area of size 1.37 times greater. But a configuration with 2 FPUs that implement FMA can issue 2 multiplications and 2 additions per cycle, the same as a configuration with 4 FPUs (configuration C) but at a lower cost (the area of B is 46.7% of the area of C).

In addition, the register file access time can force the cycle time in a VLIW architecture, and the access time is governed by the number of registers, the width of the registers and the size of each register cell (which depends on the number of read and write ports). To calculate the impact of the number of ports in the access time, we use a model [13] based on the CACTI model for cache memory [29]. Next we show the impact of the number of ports using the same configurations of the area cost example (A, B and C of Table 1). Configuration B has an access time 8% slower than configuration A, while the access time of configuration C (4 FPUs without FMA) is 42% slower than the access time of configuration A. To reduce the access time, the register file can be partitioned into several files, maintaining copies of all the data [3], but at the cost of increasing the total area cost. For example, configuration C has 10 read plus 6 write ports. This RF can be also implemented by two identical copies, where all functional units can write in both copies of the RF, but only 1 bus and 2 FPUs read each copy. In this case, 5 read plus 6 write ports are required for each copy. This implementation increases the register file area by 12.7%, but reduces the access time from 42% to 21% slower than A.

Table 1: Area cost for several 32-RF configurations.

Configuration	Ports	Area of a memory cell	Total RF area (λ^2)
A) 2 bus + 2 fpu	6R+3W	$14664\ \lambda^2$	32.08×10^6
B) 2 bus + 2 FMA fpu	8R+3W	$21420\ \lambda^2$	43.87×10^6
C) 2 bus + 4 fpu	10R+6W	$45820\ \lambda^2$	93.84×10^6

4. Performance evaluation

In this section we first show an evaluation of the performance limits that are expected from the use of replication, widening and fusion techniques. We first consider an ideal framework with a perfect scheduling and an infinite register file. Then we make real both aspects and analyse the influence of using a heuristic algorithm to do software pipelining and having a finite register file. Finally we consider implementation costs (area and cycle time) and draw a set of conclusions from a performance/cost conscious evaluation of the different implementation alternatives.

The processor configurations considered in this section are named using two different possibilities: *XwY* and *xZ* (where *Z* equals *X* times *Y*). Each one of these configurations has *X* bidirectional buses (to perform load/store operations) and twice the number of functional units[1]. The width of the resources is *Y* words. For instance, configuration 2w1 has 2 buses and 4 FPUs and all the resources have a width of 1 word, while configuration 1w2 has 1 bus and 2 FPUs, all of width 2 words. As the baseline configuration that we consider is *1w1*, both configurations can be also named *x2* because they can issue 2 times the number of operations of the baseline. The latencies are as follows: a store is served in 1 cycle; division and square root are not pipelined and require 19 and 27 cycles, respectively; the rest of the operations (load, add,...) are fully pipelined and require 4 cycles to be executed. We append *F* to those configurations in which FMA is implemented. The memory is ideal and all references always hit in the cache.

4.1 Maximum ILP achievable

First of all we analyse the performance of FMA under the following ideal situations: perfect scheduler (i.e. always achieves the *MII*) and unbounded number of registers (so there is no need to perform spill code).

1.Preliminary studies show that a relation of 2 FPUs for each bus is the most balanced configuration (see Figure 3). Also, we have based the cost calculations on the MIPS R10000, which can issue 2 floating point and 1 memory operation per cycle.

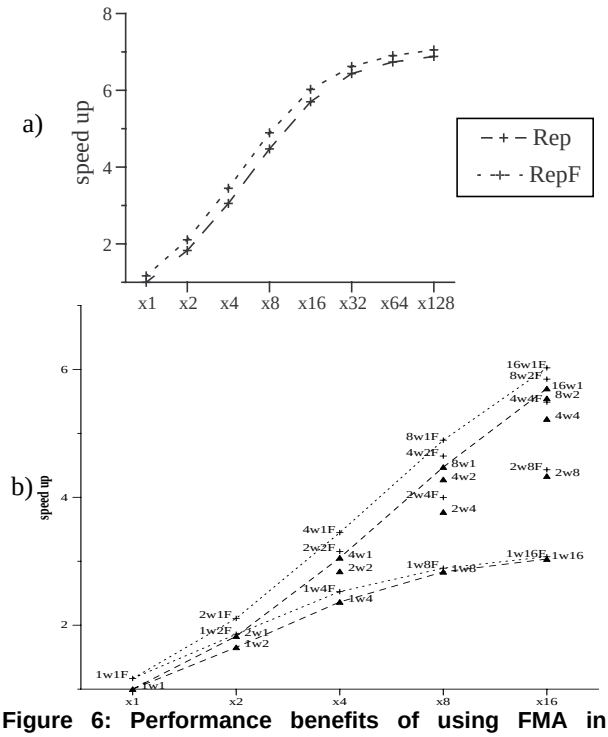

Figure 6: Performance benefits of using FMA in processor configurations based on a) replication and b) a combination of replication and widening.

Figure 6.a shows the additional performance benefits of using FMA for a set of processor configurations based on replication (*xZ* with *Zw1*). On one side, notice that the performance that is obtained does not scale with the peak performance of the architecture. This is due to the fact that aggressive configurations convert resource-bound loops into recurrence-bound loops; as a consequence, adding more resources does not contribute to a real increase in performance. On the other hand, notice that the difference between using FMA (dotted plot) or not using FMA (dashed plot) is practically a constant; this is due to the fact that the FMA operation produces benefits to recurrence-bound loops in very aggressive configurations (see Figure 5), maintaining an increase of performance similar to the obtained for compute-bound loops in configurations with a smaller degree of aggressiveness.

Figure 6.b shows the additional performance benefits of using FMA for a reduced set of processor configurations that apply combinations of both replication and widening. Notice that FMA can play an important role, specially in aggressive configurations where replication is used (for instance, configuration *1w8F* has a performance 2.1% better than *1w8*, while *8w1F* has a performance 9.4% better than *8w1*).

4.2 The effect of spill code on performance

The goal of the studied techniques is to increase the ILP of loops by reducing their Initiation Interval. Regretfully, reducing the *II* can increase the register requirements [15]. If the number of registers required to schedule a loop on an architecture exceeds the number of physical registers available, spill code must be introduced in order to free some registers [16]. However, spill code increases the memory traffic and can result in an increase of the *II*, with the associated performance degradation. We schedule the loops using the HRMS heuristic [17][18], a register pressure sensitive heuristic that achieves near optimal schedules. Register allocation is performed using the wands-only strategy and the end-fit with adjacency ordering [25].

Figure 7 shows the performance for different processor configurations and sizes of the register file (*r-RF* configuration uses a register file of *r* registers, *Y* words wide). The baseline configuration considered is *1w1* with a *256-RF*. This configuration does not require spill code and therefore is equivalent to the baseline configuration in Figure 6. Notice that the configuration *8w1* does not include the *32-RF* bar; this is because this configuration can produce 24 results per cycle (8 memory and 16 FPU), and we consider a 4-cycle latency configuration. In this case, the register pressure is so high, that the scheduler fails to produce a valid schedule with the available registers. This situation becomes more critical in configurations with a factor higher than x8, and this is the reason why we do not present the results of these configurations with this latency model. Figure 7 shows the expected results: the performance grows with the aggressiveness of configurations, and the register file size has an important impact on the final performance. The FMA operations (white portion in each bar) improves the performance of all configurations but the increase is higher for those configurations with high register pressure.

Another important point can be observed in this Figure: there is a significative difference between the performance reported in this section, and the theoretical performance shown in section 4.1. For instance, there is an increase of the theoretical performance of 7.5% from configuration 8w1 to configuration 8w1F; when the loop is scheduled with a 64-register file, this difference grows up to 20.5%. This difference is due to other factors that have an important contribution on the final performance. In this point we are going to analyse these factors.

The total amount of cycles required to execute a loop in a given architecture can be divided into three components:

- the *MII*: this is the minimum number of cycles required to execute a loop; it depends on the characteristics of the loop itself and on the architecture.
- cycles due to spill code: these are the cycles due to the code introduced to fit the scheduling in the available number of registers.
- and cycles due to the scheduler heuristic: these are the cycles added by the scheduler when it fails in finding the optimal scheduling.

Figure 8 shows, for the same processor configurations, the distribution of cycles introduced by each of these components. Each plot is divided in 8 columns grouped in pairs representing the register file size (horizontal axis): the

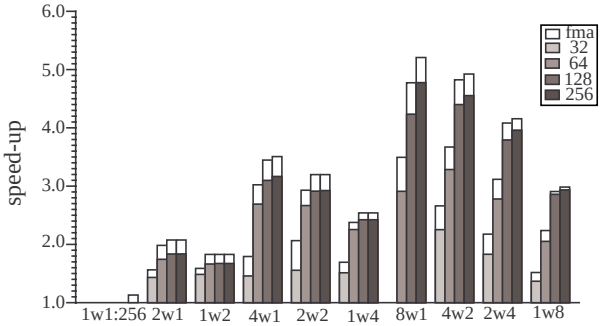

Figure 7: Performance of some processor configurations for several sizes of the register file. Baseline: configuration 1w1 with a 256-RF

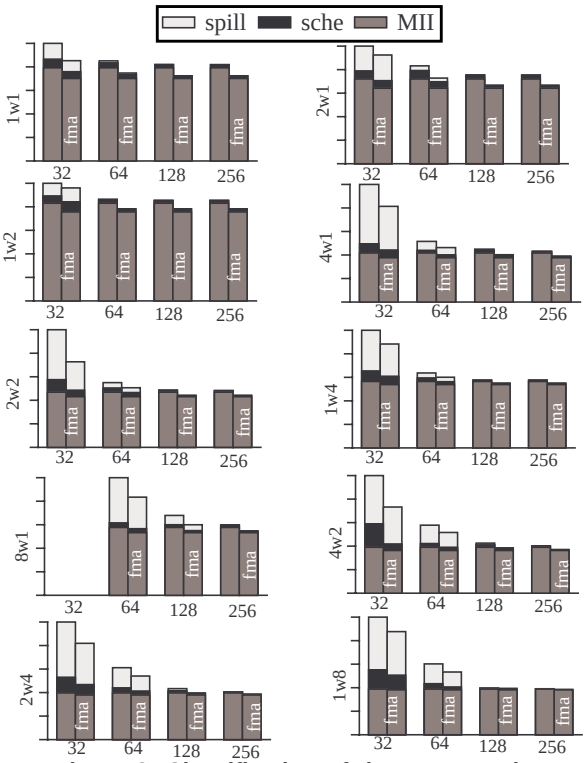

Figure 8: Classification of the spent cycles

right column in each pair represents an architecture with FMA and the left column an architecture without FMA. The use of FMA reduces the cycles required for all the factors previously defined:

- the dark gray part represents the minimum theoretical cycles, so it is independent of the register file size. Notice that using FMA reduces the MII because some operations have been fused (and there are less operations to be scheduled). For instance, the number of cycles is reduced in 10.7% when FMA is used in configuration *4w1*.
- the light gray part shows the spill code cycles. FMA reduces the register pressure of the loops because it does not require a register to store the intermediate result (e.g. the number of cycles due to spill code is reduced in 35.7% when FMA is used in configuration *4w1* with a 32-RF).
- the black part represents the scheduler cycles. The reduction of the number of operations to schedule (due to the reduction of operations in the loop and to the reduction in the spill code required) makes the graph less complex; therefore, the scheduler has more opportunities to find a schedule closer to the optimal. For example, the number of cycles added by the scheduler is reduced in 20.9% when FMA is used in configuration *4w1* with a 32-RF.

From the analysis of Figure 8, one can conclude that using FMA has an impact on the final performance greater than the performance that we can expect if we only take into account the theoretical analysis. For instance, configuration *4w1F* with a 32-RF has a performance 22.9% better than configuration *4w1*, while the difference in the theoretical analysis was 10.7%.

4.3 Performance/cost trade-offs

In this section we try to identify the role of replication, widening and FMA when their implementation is taken into consideration. Area and cycle time cost can easily offset any gain in ILP or even worse, make a configuration non implementable. The 1994 *Semiconductor Industry Association* (SIA) predictions [26] are used to define a set of future technology generations and their characteristics.

Table 2: SIA predictions in 1994

	1998	2001	2004	2007	2010
λ (μm)	0.25	0.18	0.13	0.10	0.07
Size (mm^2)	300	360	430	520	620
λ^2 per chip ($x10^6$)	4800	11111	25443	52000	126530
λ^2 / mm^2 ($x10^6$)	16	30.86	59.17	100	204.08

We study a broad set of architectures that implements the replication, widening and fusion techniques. In order to perform a realistic study, the register files of the configurations have been partitioned to reduce their access time (and therefore, the configurations cycle time). Configurations are labelled *XwY(Z:n)* (i.e. *X* buses and *2*X* FPUs, all of width *Y*, with a RF of *Z* registers of width *Y*, partitioned in *n*-blocks) or *XwYF(Z:n)* (i.e. the same but the FPUs can implement the FMA operation).

The methodology used for this evaluation is as follows. First we calculate the cost of the tested configurations and decide which is implementable for each technology generation (we consider that a configuration is implementable for a technology generation if its FPUs area plus its register file area do not exceed 20% of the total chip area). Second, for each implementable configuration we calculate its cycle time, assuming that the cycle time is defined by the register file access time. Each FPU requires an amount of time to perform one operation; its latency in cycles depends on the processor cycle time. The cycle models we have used are listed in Table 3 and assume that the baseline configuration (*1w1*) uses the 4-cycles model.

Table 3: Cycle models.

cycle model	latency (in cycles)			
	store	+,*, load	div	sqrt
4-cycles	1	4	19	27
3-cycles	1	3	15	21
2-cycles	1	2	10	14
1-cycle	1	1	5	7

Other configurations will fit into the appropriate model depending on its relative (from the *1w1* configuration) cycle time. A configuration with a relative cycle time Tc belongs to the z-cycles model, where $z = \lceil 4/Tc \rceil$. Finally, we perform the scheduling to find the cycles required[1]. The cycles required to execute all the loops times the cycle time give us the final performance. In all cases, we use a fixed timing model based on technology parameters for λ=0.25. We have not attempted to factor-in reductions in cycle time due to future technology generations.

Before going into the final results, let us analyse the individual effects of some parameters on the configurations evaluated:

- Number of registers. Having large register files reduces the register pressure and the need of spill code. However, the increase in the cycle time may counteract this gain. For example, Figure 9.a shows the performance/cost ratio

1.The cycles required are calculated as the cycles per iteration (Initiation Interval) times the number of iterations performed in the original loop execution.

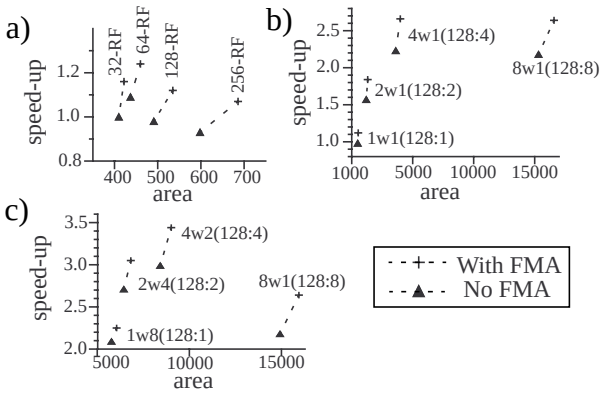

Figure 9: effect of a) increasing the RF size, b) replication and c) different ways to implement a configuration with the same peak performance. All figures compare FPUs with and without FMA.

when we increase the number of registers available in the register file, for configuration 1w1. Notice that the performance for this configuration declines when we use a register file larger than 64. This configuration has negligible need for spill code when a 64-RF (or bigger) is available, so an increase of the registers file does not affect the cycles required, but increases the cycle time.

- Replication. Configurations based on replication report good increases in ILP. However, high degrees of replication can make the configuration unimplementable (they occupy more than 20% of the total chip area) or suffer a decrease in performance because a small increase in IPC (instructions per cycle) is counteracted with a high increase of the cycle time. For instance, Figure 9.b shows the performance/cost ratio when only replication is applied.
- Replication and widening. The same peak performance can be obtained by applying different degrees of replication and widening. Although replication is more versatile and reported higher ILP returns, cycle time puts configurations based on small degrees of widening in a better position, as shown in Figure 9.c for *x8* configurations.
- Fused Multiply-add. FMA returns good performance relative to its low implementation cost. The three plots in Figure 9 also shows the performance of the same configurations when FMA is included (the cross marks). For instance, configuration *8w1F(128:8)* performs 21.1% better than *8w1(128:8)*, with an increment of only 8.3% in the area. Also, configuration *2w4F(128:2)* has a performance 2% better than configuration *4w2(128:4)* and only has 72% of its area requirements.

Assuming the SIA predictions and our area cost models, Figure 10 shows the area cost for a broad range of processor configurations that include replication, widening and FMA with different sizes for the RF (notice that the vertical axis has logarithmic scale). Each horizontal line represents, each technology generation, the 20% in area devoted to implement this processor core and therefore defines the set of implementable configurations for this technology.

Figure 11 shows, for each technology, the five configurations that achieve the best performance. First of all, notice that none of the most aggressive configurations are in the top-five configurations due to their high cost: the configurations that offer best performance are the ones that combine small degrees of replication and widening. For each technology, we have also highlighted the "eligible"

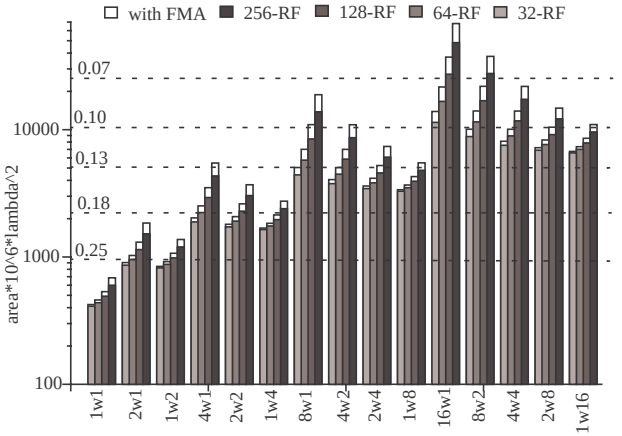

Figure 10: Implementable configurations for each tehcnology generation considered.

configurations (black triangles). A configuration is "eligible" if there is not any configuration that can achieve the same performance, or better, with a small cost. Notice that all except for two "eligible" configurations implement the FMA operation in their FPUs. For example, for a technology of λ=0.13, the configuration with best performance is *2w4(128:2)*, using 18.7% of the total chip area. The configuration with the second best performance is *2w2F(64:2)* that achieves 99.3% of the performance of the first one, using only 8.18% of the total chip area. Configuration *2w4F(128:2)* is not included in the plot because it requires 20.6% of the total chip area (more than 20%) but offers a performance 12.5% greater than 2w4(128:2).

We can conclude that using FPUs that implement the FMA operation has some costs, but the benefits that offers

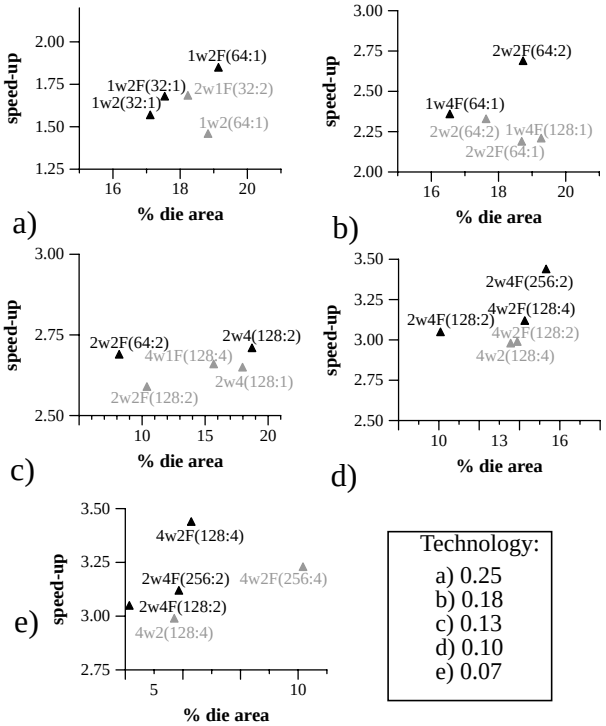

Figure 11: Top five configurations. The increment of the clock speed has not been taken into account

overcome these costs, so it is a good solution to increase the performance of aggressive configurations.

5. Conclusions

In order to exploit the inherent ILP of numerical applications, aggressive processors are required. More operations can be executed per cycle by either increasing the number of functional units (replication technique) or by increasing the number of data each functional unit can process per cycle (widening technique). However, the more aggressive the processor more critical become the recurrences. Fused multiply-add functional units increase the number of operations performed by cycle and improve performance in recurrence-bound loops that contain multiply-add chains in their critical recurrence.

In this paper we have evaluated the impact on performance and cost of FMA functional units in aggressive architectures. In particular we have evaluated the influence of FMA units in combination with the widening and replication techniques. The evaluations have been performed over a large number of software pipelined loops from the Perfect Club benchmarks assuming a VLIW architecture.

We have analysed the effects of FMA on resource-bound loops and on recurrence-bound loops. We have studied the ILP limits of each configuration in optimal conditions, showing that FMA units provide a significant advantage. This is because FMA units increase the peak number operations that can be performed per cycle and can reduce the latency of critical recurrences. When a limited number of registers is considered, the advantage of using FMA increases. This increment in performance is due to two reasons: the influence of spill code (the register requirements are reduced) and the influence of the scheduler (it has a simpler task since there are less operations to schedule).

Taking into account that using FMA units is more expensive in terms of area cost and cycle time, we have estimated the cost of the configurations considered. We compare the performance of the configurations that can be built for the next processor generations. The performance has been calculated using the register file cycle time. To perform a realistic comparison, the register file cycle time has been reduced using the partitioning technique and the FPUs latency has been adapted to the cycle time. From this study we conclude that, for a given technology, the best performance is obtained for configurations that use FMA units requiring less area.

Acknowledges

This work has been supported by the Ministry of Culture and Education of Spain under contract TIC 98-0511 and by CEPBA (European Center for Parallelism of Barcelona).

References

[1] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero. “Ictíneo: A tool for Instruction-Level Parallelism Research”. Tec. Rep. UPC-DAC-1996-61 U. Politècnica Catalunya. Dec 1996.

[2] M. Berry, D. Chen, P. Koss and D. Kuck. “The Perfect Club benchmarks: Effective performance evaluation of supercomputers”. Tec. Rep. 827, CSRD-U. Illinois at Urbana-Champain, Nov. 1988.

[3] A. Capitanio, N. Dutt and A. Nicolau. “Partitioned register files for VLIWs: A preliminary analysis of tradeoffs”. In *Proc. MICRO-25*, pp 292-300, Dec. 1992.

[4] J.C. Dehnert and R.A. Towle. “Compiling for Cydra 5”. In *Jour. Supercomputing*, vol. 7 no. 1/2, pp 181-228, May 1993.

[5] W.-M. W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm and D.M. Lavery. “The superblock: An effective technique for VLIW and superscalar compilation”. In *Jour. Supercomputing*, vol. 7 no. 1/2, pp 229-248, May 1993.

[6] P.Y.T. Hsu. “Design of the TFP microprocessor”. *IEEE Micro*, vol. 14 no. 2 pp 23-33. April 1994.

[7] IBM. Special issue on the RS/6000. *IBM Jour. Res. and Develop*. vol. 34 no. 1. January 1990.

[8] R. Jolly. “A 9-ns 1.4 gigabyte 17-ported CMOS register file”. *J. of Solid-State Circuits*, vol. 25 no. 10 pp 1407-1412, Oct. 1991.

[9] R.M. Jessani and M. Putrino. “Comparison of single- and dual-pass multiply-add fused floating-point units”. In *IEEE Trans. on Computers*, vol. 47 no. 9, pp 927-937, Sep. 1998.

[10]M. Lam. “Software pipelining: An effective scheduling technique for VLIW machines”. In *Proc. PLDI-88*, pp. 318-328, June 1988.

[11]C. G. Lee. “Code Optimizers and Register Organizations for Vector Architectures”. Ph.D. thesis U. C. Berkeley. May 1992.

[12]D. López, J. Llosa, M. Valero and E. Ayguadé. “Resource widening vs. replication: Limits and performance-cost trade-off”. In *Proc. ICS-12*, pp 441-448. July 1998.

[13]D. López, J. Llosa, M. Valero and E. Ayguadé. “Widening Resources: A Cost-effective Technique for Aggressive ILP Architectures”. In *Proc. MICRO-31* pp 237-246. Nov.-Dec. 1998.

[14]D. López, M. Valero, J. Llosa and E. Ayguadé. “Increasing memory bandwidth with wide buses: Compiler, hardware and performance trade-off”. In *Proc. ICS-11*, pp 12-19. July 1997.

[15]J. Llosa, E. Ayguadé and M. Valero. “Quantitative evaluation of register pressure on software pipelined loops”. In *Jour. of Parallel Programming*, vol. 26 n. 2 pp. 121-142. 1998

[16]J. Llosa, M. Valero and E. Ayguadé. “Heuristics for Register-Constrained Software Pipelining”. In *Proc. MICRO-29*, pp. 250-261, Dec. 1996.

[17]J. Llosa, M. Valero, E. Ayguadé and A. González. “Hypernode Reduction Modulo Scheduling”. In *Proc. MICRO-28*, pp 350-360, Dec. 1995.

[18]J. Llosa, M. Valero, E. Ayguadé and A. González. “Modulo Scheduling with reduced register pressure”. In *IEEE Trans. on Computers*, vol. 47 no. 6 pp 625-638, June 1998.

[19]Microprocessor Report vol 11, no. 14. *Intel HP make EPIC disclosure*. October 1997.

[20]Microprocessor Report vol 12, no. 6. *AltiVec vectorizes PowerPC*. May 1998.

[21]Microprocessor Report vol 12, no. 12. *TI aims for floating-point DSP lead*. September 1998.

[22]Microprocessor Report vol 12, no. 16. *MAP1000 unfolds at Equator*. December 1998.

[23]B.R. Rau. “Iterative modulo scheduling: An algorithm for software pipelining loops”. In *Proc. MICRO-27* pp.63-74, Nov. 1994.

[24]B.R. Rau and C.D. Glaeser. “Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing”. In *Proc. 14th Ann. Microprogramming Workshop*, pp. 183-197, October 1981.

[25]B.R. Rau, M. Lee, P. Tirumalai, and P Schlansker. “Register allocation for software pipelined loops”. In *Proc. Progr. Lang. Des. and Impl. (PLDI-92)*, pp. 283-299, June 1992.

[26]Semiconductor Industry Assoc. The National Technology Roadmap for Semiconductors. SIA , San Jose, California 1994.

[27]T. Watanabe. “The NEC SX-3 supercomputer system”. In *CompCon91* pp. 303-308, 1991

[28]S.W.White and S. Dhawan. “POWER2: Next Generation of the RISC System/6000 family”. *IBM Jour. Res. Develop.* Vol 38 no 5, pp 493-502. September 1994.

[29]S.J.E. Wilton and N.P. Jouppi. “CACTI: An enhanced cache access and cycle time model”. *IEEE Jour. of Solid-State Circuits*, Vol. 31 no 5, pp 677-688, May 1996.

[30]K.C. Yeager. “The MIPS R10000 superscalar microprocessor”. *IEEE Micro* v. 16 n. 2 pp. 28-40, March 1996.