Cost-Conscious Strategies to Increase Performance of Numerical Programs on Aggressive VLIW Architectures

David López, Josep Llosa, Mateo Valero, *Fellow*, *IEEE*, and Eduard Ayguadé

Abstract—Loops are the main time-consuming part of numerical applications. The performance of the loops is limited either by the resources offered by the architecture or by recurrences in the computation. To execute more operations per cycle, current processors are designed with growing degrees of resource replication (*replication technique*) for memory ports and functional units. However, the high cost in terms of area and cycle time of this technique precludes the use of high degrees of replication. High values for the cycle time may clearly offset any gain in terms of number of execution cycles. High values for the area may lead to an unimplementable configuration. An alternative to resource replication is resource widening (*widening technique*), which has also been used in some recent designs in which the width of the resources is increased (i.e., a single operation is performed over multiple data). Moreover, several general-purpose superscalar microprocessors have been implemented with multiply-add fused floating-point units (*fusion technique*), which reduces the latency of the combined operation and the number of resources used. In this paper, we evaluate a broad set of VLIW processor design alternatives that combine the three techniques. We perform a technological projection for the next processor generations in order to foresee the possible implementable alternatives. From this study, we conclude that if the cost is taken into account, combining certain degrees of replication and widening in the hardware resources is more effective than applying only replication. Also, we confirm that multiply-add fused units will have a significant impact in raising the performance of future processors architectures with a reasonable increase in cost.

Index Terms—VLIW processors, instruction level parallelism, software pipelining, numerical applications, performance/cost trade-off.

◆

1 INTRODUCTION

Current high-performance microprocessors rely on hardware and software techniques to exploit the instruction-level parallelism (ILP) available in applications. These processors make use of deep pipelines in order to reduce the cycle time and wide instruction issue units to increase the number of instructions executed per cycle. The selection of the instructions issued each cycle is done either at runtime in out-of-order superscalar processors or at compile time in Very Long Instruction Word (VLIW) architectures.

In a basic VLIW architecture, an instruction is composed of a number of operations that are issued simultaneously to the functional units. For these architectures, the compiler is responsible for the scheduling of the operations, so the dispatch phase is very simple. These architectures have been widely used in the DSP arena (as in Texas Instruments' 'C6701 [27] and Equator's MAP1000 [28]) and constitute the core of several current designs (as Sun's MAJC [29], [31] and the EPIC from Intel-HP [25], [30]) targeting general-purpose applications.

The static nature of VLIW schedulings requires good compilation techniques to effectively exploit the ILP available in real programs [34]. Loops are the main time-consuming part of numerical programs. Software pipelining [1] is a compilation technique that extracts ILP for the innermost loops by overlapping the execution of several consecutive iterations. Modulo scheduling [35] is a class of software pipelining algorithms that has been incorporated in many production compilers. In a modulo-scheduled loop, the *Initiation Interval* is the number of cycles between the initiation of two consecutive iterations. The Initiation Interval is bounded either by the recurrences in the dependence graph or by the resource constrains of the target architecture [7], [33].

The performance of loops bounded by resources can be improved by increasing the number of resources available in the architecture (*replication technique*). Using this technique, the number of operations that can be simultaneously executed over independent data is increased. As an alternative to replication, the width of the resources can be increased, exploiting data parallelism at the functional unit level, like in vector processors, or, in superscalar and VLIW processors, using the *widening technique*. Using this technique, a single operation can be performed over multiple data (SIMD). Both replication and widening can be combined in the same processor design. Each one of the configurations has a different performance, but also a different cost.

The use of replication and widening allows us to have a scalable architecture in which hardware to increase the number of operations performed per cycle can be added. Although replication has been extensively used in the design of superscalar processors, only small degrees of

---

- *The authors are with the Computer Architecture Department, Technical University of Catalunya (UPC), Campus Nord, Modul D6, Jordi Girona 1-3, 08034-Barcelona, Spain.*
E-mail: {david, josepell, mateo, eduard}@ac.upc.es.

Manuscript received 21 Apr. 2000; revised 5 Feb. 2001; accepted 4 May 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111976.

0018-9340/01/$10.00 © 2001 IEEE

widening have been applied in current microprocessors. For instance, IBM's POWER2 [39] applies widening to the memory ports. Vector processors, like NEC's SX-3 [38], apply widening to the floating point units (FPUs). Multimedia processors extensions combine sub and superword parallelism, which, in some way, use the basic idea of widening applied to integer and FP operations, like the Motorola AltiVec [26] or the MultiMedia eXtensions (MMX) and Stream SIMD eXtensions (SSE) in Pentium III [12].

As the number of transistors on a single chip continues to grow, more hardware can be accommodated on a chip, so future microprocessors will use these techniques to exploit ILP aggressively. Replication and widening focus on increasing the performance of resource-bound loops. Consequently, the more aggressive the architectures, the more critical the recurrences. Several current microprocessors use functional units that can perform multiple elemental operations as a monolithic complex operation; for instance, *fused multiply and add* (FMA) FPUs perform a multiplication and a dependent addition as a single operation. This technique (*fusion technique*) has been implemented in the floating-point units of several microprocessors, such as the IBM RS/6000 [11], IBM POWER2 [39], MIPS R8000 [10], and MIPS R10000 [42], and can reduce the latency of the recurrences, but also has some advantages for resource-bound loops.

This paper focuses on a cost-conscious evaluation of a broad range of VLIW processor designs in which the replication, widening, and fusion techniques are combined. The paper targets numerical applications based on FP operations. Here, we extend and improve the previous research work done in [22], [23], and [24]. In this evaluation, we take into account the individual impact of the static scheduler, register file size, area, and cycle time. The analysis of the results coming from this evaluation obliges us to view the performance from a global perspective.

The area cost defines those configurations that could be implemented in the next microprocessor generations, according to the predictions of the *Semiconductor Industry Association* [37]. 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, for a given technology, the best performance is obtained when replication, widening, and fusion are appropriately combined. Related studies have been conducted in the context of custom-fit processors [9] and transport triggered architectures [13].

All the evaluations have been performed for VLIW architectures and numerical programs. Our workbench is composed of 1,180 loops that account for 78 percent of the execution time of the Perfect Club benchmarks [5]. The loops have been obtained using the experimental tool Ictíneo [4] and software pipelined using *Hypernode Reduction Modulo Scheduling* [21]. Register allocation has been performed using the wands-only strategy and the end-fit with adjacency ordering [36]. When a loop requires more than the available number of registers, spill code is added and the loop is rescheduled [20]. The focus of this paper is to perform a study of architectural techniques that can be used to improve ILP, assuming a given compilation technology. This is the reason why we do not analyze the impact of compiler transformations (such as loop interchange, unroll-and-jam, tiling, etc.) and refined techniques to analyze dependencies [41]. These transformations may lead to innermost loops that are able to take even more advantage from processor core organizations which include some degrees of widening. We believe that the general conclusions of the paper will not change and provide the same insights for microprocessor designers.

The organization of the paper is as follows: Section 2 presents the basic concepts of modulo scheduling and the issues that limit the performance (for instance, register pressure). Section 3 describes the techniques evaluated in this paper (replication, widening, and fusion) and includes some examples to understand their role in improving the exploitation of ILP. This section ends with a classification of loops to give an idea of the potential benefit of the described techniques in the final performance. Section 4 presents the workbench following the previous classification. In Section 5, we present some design considerations (area cost, register file access time) which are the main ingredients of the cost-conscious evaluation performed in Section 6. Finally, Section 7 summarizes the main conclusions of this work.

2 OVERVIEW OF RELATED CONCEPTS

This section includes an overview of software pipelining [1], modulo scheduling [35], and the problems caused by the high register pressure introduced by these aggressive techniques [19].

2.1 Data Dependence Graph (DDG) and Extended Data Dependence Graph (EDDG)

The dependences of an innermost loop can be represented by a Data Dependence Graph:

Definition 1: Data Dependence Graph $G = DDG(V, E, \delta)$**.** *V is the set of vertices, where each vertex $v \in V$ represents an operation of the loop body. E is the dependence edge set, where each edge $(u, v) \in E$ represents a dependence between two operations u and v. The dependence distance $\delta_{(u,v)}$ is a nonnegative integer associated with each edge $(u, v) \in E$. There is a dependence with distance $\delta_{(u,v)}$ between two nodes u and v if the execution of the operation v depends on the execution of operation u, $\delta_{(u,v)}$ iterations before.*

One of the studied techniques is the widening technique, which compacts memory and arithmetic operations (see Section 3.1). The compaction of memory operations is based on accesses to data stored in consecutive memory locations. For this reason, we define the Extended Dependence Graph that includes information about strides:

Definition 2: Extended Dependence Graph $G' = EDG(G, S, \sigma)$. *G is the dependence graph defined in Definition 1. There is a stride edge $(u, v) \in S$ if u and v represent the same kind of memory access (load or store) to the same array and there is a constant difference between the addresses accessed in operations u and v. This difference is the stride $\sigma_{(u,v)}$ associated to the edge (u, v).*

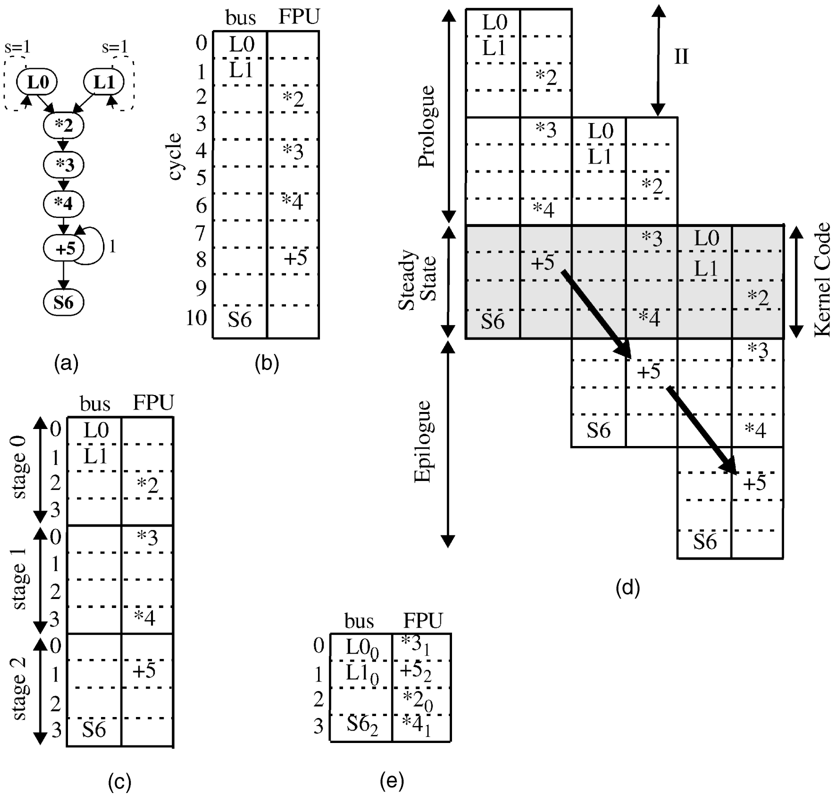


Fig. 1. (a) The sample EDDG, (b) one possible scheduling, (c) scheduling of one iteration after applying software pipelining, (d) software pipelining loop execution, and (e) kernel code.

For example, assume the EDDG in Fig. 1a. It consists of seven nodes, labeled *0* through *6*, that represent basic operations: three memory operations (loads *L0*, *L1*, and store *S6*) and four arithmetic operations (products **2*, **3*, and **4*, and addition *+5*). The dotted edges represent the stride between memory operations (e.g., load L0 has a stride of 1 with itself). The solid edges between nodes represent values generated and consumed by these operations. All the solid edges have distance zero except for edge $< +5, +5 >$, which has a distance of one iteration: $+5_{n+1}$ (i.e., node +5 of iteration $\mathbf{n+1}$) uses the data generated by $+5_n$ (i.e., the same node in the previous iteration). This dependence defines a recurrence (cycle in the dependence graph).

2.2 Software Pipelining and Modulo Scheduling

Assuming that memory operations and arithmetic operations can be served in one and two cycles, respectively, Fig. 1b shows one possible scheduling for an architecture whose resources are one memory port (bus) and one floating point unit (FPU). In this case, 11 cycles per iteration are required.

In a software pipelined loop, the scheduling of an iteration is divided into stages so that the execution of consecutive iterations that are in distinct stages is overlapped. There are different approaches to generate a software pipelined scheduling for a loop. Modulo scheduling is a class of software pipelining algorithms that was proposed at the beginning of the 80s [35] and has been incorporated in many production compilers. The number of cycles per stage is termed *Initiation Interval (II)*. The execution of a loop can be divided into three phases: a ramp up phase that fills the software pipeline, a steady state phase where the same pattern of operations is executed in every stage and the software pipeline achieves maximum overlap of iterations, and a ramp down phase that drains the software pipeline. During the steady state phase, the execution of a new iteration starts every *II* cycles. This Initiation Interval *II* between two successive iterations is bounded either by loop-carried dependences in the graph (*RecMII*) or by the resources constrains of the architecture (*ResMII*). This lower bound on the *II* is termed the Minimum Initiation Interval ($MII = max$ (*ResMII*, *RecMII*)). Sometimes, unrolling is required to match the number of resources required by the loop with the resources of the processor and also to schedule loops with a fractional *MII* [16].

In the loop of Fig. 1a, $ResMII = 4$ because there are four FPU operations and only one FPU unit. The recurrence forces us to schedule node $+5_{n+1}$ two cycles after node $+5_n$, so $RecMII = 2$ cycles and $MII = max(4, 2) = 4$. Loops constrained by the resources are know as *resource-bound* loops, while loops whose performance is limited by the recurrences are know as *recurrence-bound* loops. Fig. 1c shows the schedule of this loop divided into three stages of four cycles. Notice that the loop has been rescheduled (nodes **4*, *+5*, and *S6* have been delayed one cycle with respect to the linear schedule of Fig. 1b) in order to make possible the iteration overlap. Fig. 1d shows the execution of three iterations, where the data dependence due to the recurrence is marked by an arrow. In Fig. 1d, it can be observed that a new iteration starts every II cycles by

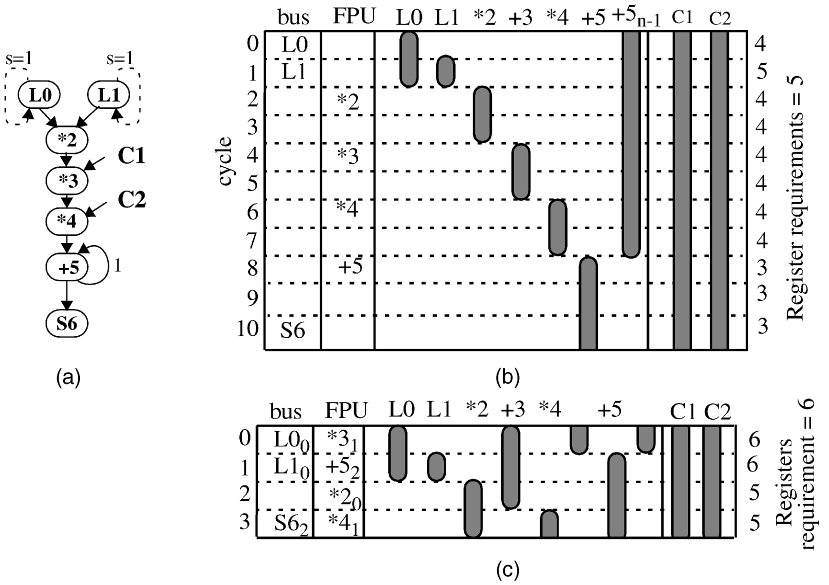


Fig. 2. (a) Sample graph and register requirements, (b) before and (c) after applying software pipelining.

overlapping different iterations, generating a pattern of length II cycles that is indefinitely repeated. This pattern is shown in Fig. 1e, where all nodes have a subscript that indicates the stage to which the node belongs.

2.3 Register Pressure

The register requirements for a given schedule depend both on the number of loop-invariants and on the *lifetime* of the variables. We assume that the lifetime of a loop variant starts when the producer is issued and ends when the last consumer is issued; for instance, in the graph of Fig. 2a the data produced by the load *L0* has a lifetime of two cycles from the moment in which the load is issued (cycle 0) through the moment the last consumer (*2) is issued (cycle 2). The minimum number of registers is estimated as the maximum number of variables alive for all cycles in the schedule (*MaxLive*). In the example of Fig. 2b, $MaxLive = 3$ because three variables are alive in the same cycle (cycle 1): two from loads L0 and L1 and another one containing the result of *+5* of the previous iteration. As shown in [36], the maximum number of simultaneously live values is an accurate approximation of the number of registers required by the schedule.

There are also two loop-invariants in the loop (constants C1 and C2, right part of Fig. 2b). Loop-invariants are repeatedly used, but never defined, during loop execution and have a single value during all the iterations of the loop. Therefore, they require one register each, regardless of the schedule and the machine configuration. So, in the schedule of Fig. 2b, five registers are required (two for the loop-invariants and three for the loop-variants).

Fig. 2c shows the register requirements after applying software pipelining. In that case, the loop-variant requirements increase up to four registers (in cycles 0 and 1), so the final requirements are six registers. This example shows the typical situation when software pipelining is applied: The cycles per iteration are reduced (from 11 to four in the steady state), but the register requirements increase (from five to six).

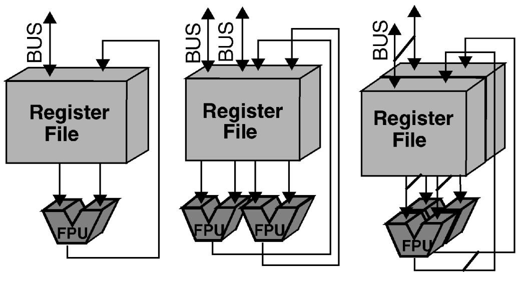
The actual number of registers required is not known until the register allocation step of the whole scheduling algorithm is done. In this work, scheduling is performed using *Hypernode Reduction Modulo Scheduling* (HRMS) [21], a register sensitive heuristic that achieves schedules with II = MII in 97.5 percent of the tested loops with reduced register requirements. To allocate registers, we use the wands-only strategy using end-fit with adjacency ordering [36], which almost never requires more than $MaxLive + 1$ registers. If the number of required registers overpass the number of physical registers available, spill code is added in order to free registers, using the iterative algorithm described in [20]. However, this additional code may reduce the performance.

3 TECHNIQUES TO INCREASE ILP AND THEIR IMPLICATIONS

3.1 Replication and Widening

In order to increase the number of operations performed per cycle, the resources of the processor must be increased. In this section, we analyze two alternatives to make the architecture more aggressive: resource *replication* and *widening*. Resource replication consists of increasing the number of resources available by adding more independent functional units. Widening consists of increasing the number of operations that each functional unit can simultaneously perform per cycle (i.e., functional units that operate in a SIMD way).

Fig. 3 shows the use of replication and widening, both for memory ports (buses) and FPUs. Fig. 3a shows a base configuration with one bidirectional bus and one 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, Fig. 3b); 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 the memory bus and the FPU



(a) (b) (c)

Fig. 3. (a) Base configuration, (b) the same configuration after applying replication, and (c) after applying widening. Notice that the register file has also been widened.

(widening technique, Fig. 3c); in this case, two consecutive words in memory can be accessed and stored in a single register (of width 2) and one operation can be performed over registers of width 2.

Fig. 4 illustrates the factors that limit the exploitation of parallelism when replication and/or widening are used. Fig. 4b shows the example loop unrolled so that two consecutive iterations of the original loop are considered for scheduling. The subscripts indicate the iteration to which the operations belong to. Let us assume that the loop in Fig. 4b is scheduled for a machine with one FPU and one bidirectional bus. Let us assume that memory operations and arithmetic operations are fully pipelined with a latency of one and two cycles, respectively. For any pair loop/architecture, it is possible to compute the bounds of the execution time of an iteration. In our example, we consider three separate bounds (see Fig. 4d1):

- Recurrences: In the loop in Fig. 4b, there is one recurrence $(+5_0, +5_1)$ that limits the execution time of the loop to four cycles per iteration of the unrolled loop since the sum of latencies is 4.
- Memory bandwidth (BUS): In the loop in Fig. 4b, there are six memory operations ($L0_0$, $L0_1$, $L1_0$, $L1_1$, $S6_0$, and $S6_1$). Since all of them are fully pipelined and we have only one memory unit, at least six cycles are required to execute them.
- Execution bandwidth (FPU): In the loop in Fig. 4b, there are eight floating-point operations ($*2_0$, $*2_1$, $*3_0$, $*3_1$, $*4_0$, $*4_1$, $+5_0$, and $+5_1$). Since all of them are fully

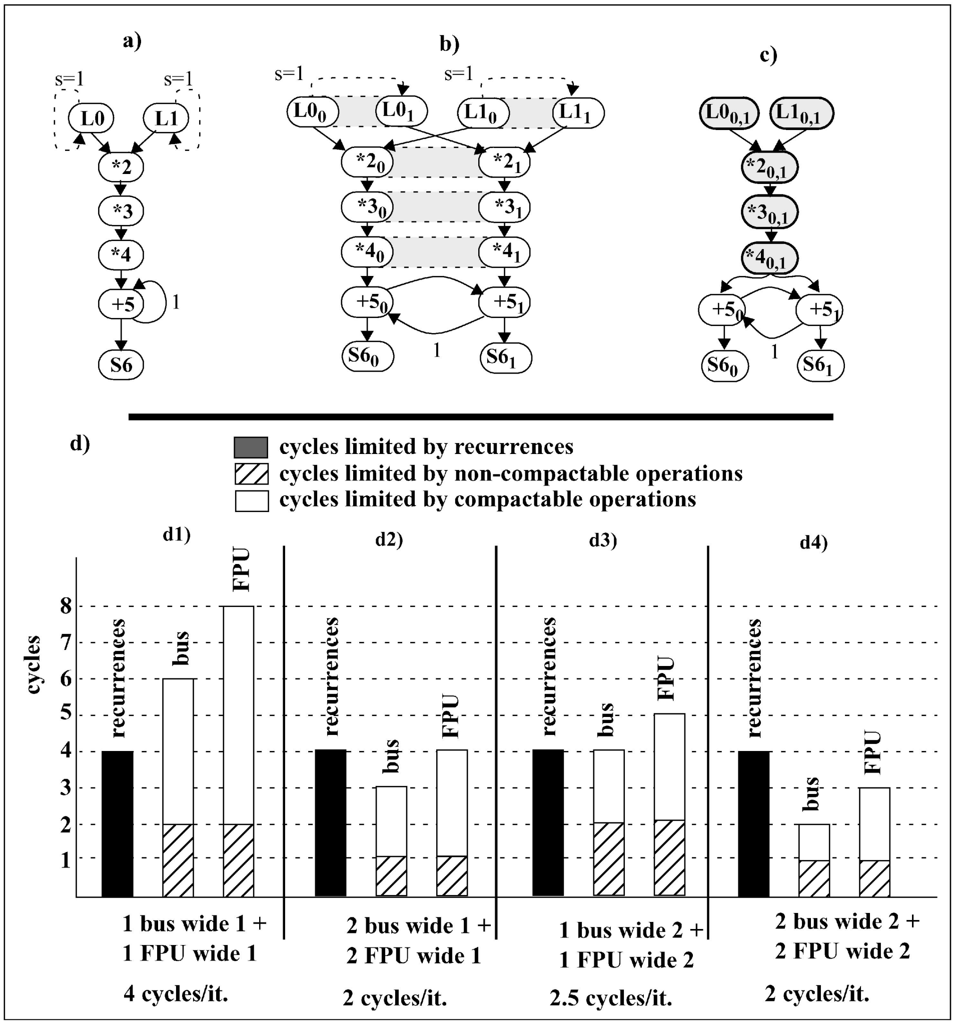


Fig. 4. (a) Dependence graph for the original loop, (b) graph after unrolling with $u = 2$ (gray shadow indicates compactable operations), (c) graph with compacted operations, (d) cycles to execute two iterations for different architecture configurations.

pipelined and we only have one FPU, at least eight cycles are required to execute them.

The most limiting factor of our example is the execution bandwidth that limits the performance to eight cycles per iteration of the loop in Fig. 4b (i.e., four cycles per iteration of the original loop).

Fig. 4b shows, with a gray shadow, the pairs of operations that can be compacted into a single wide operation. In Fig. 4d, we identify, for the different resources, the cycles contributed by compactable operations (white) and the cycles contributed by noncompactable operations (striped). In this loop, all operations are compactable except the pair $(+5_0, +5_1)$ that are inside a recurrence and, therefore, must be executed sequentially and the pair $(S6_0, S6_1)$ that cannot be stored with a wide bus because they have a stride different than one. Fig. 4c shows the loop once operations are compacted. This loop has five wide operations and four single operations.

Fig. 4d2 shows the execution bounds when the loop in Fig. 4b is scheduled for a machine where a degree of replication equals 2 has been applied (i.e., two buses and two FPUs). Notice that, in this case, we can reduce both cycles due to compactable operations and cycles due to noncompactable operations. In this case, we require four cycles to execute the eight floating-point operations. Notice that, in this case, we have reached the limit imposed by the recurrence and no more parallelism can be exploited by simply adding more resources to the architecture.

Fig. 4d3 shows the execution bounds when the loop in Fig. 4c is scheduled for a machine where a degree of widening equal to 2 has been applied (i.e., one bus and one FPU, both of width two). Notice that the cycles due to the recurrence are still four. However, the cycles due to resources have been reduced to four cycles for the memory operations and five cycles for the FP operations, summing up five cycles per iteration of the unrolled loop. However, cycles due to noncompactable operations are not reduced by widening the functional units.

Fig. 4d4 shows the execution bounds when both techniques (replication and widening) are applied. Now, we have a machine with two wide buses and two wide FPUs, therefore, we require two cycles to execute the four memory operations and 2.5 = 3 cycles to execute five floating point operations. If this loop had no recurrences, it would benefit from this reduction in cycles due to resources. However, the recurrence imposes a hard limit that cannot be reduced by adding more resources (either adding functional units or widening them).

Some preliminary conclusions can be drawn from this simple example. They will help us to understand the classification of the loops done in Section 3.3:

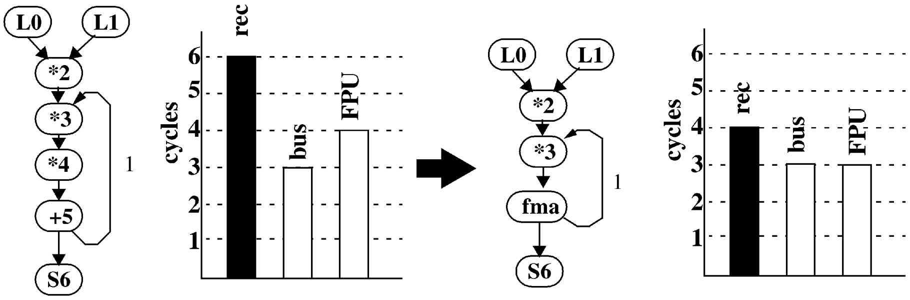
- Loops with recurrences are limited by the latency of the operations involved in the recurrence, even with an unbounded number of resources.
- Replicating the functional units reduces the limit imposed by resources by a factor equal to the replication degree.
- Widening functional units also reduces the limit imposed by resources. However, this reduction only affects the cycles contributed by compactable operations. Therefore, the noncompactable operations limit the parallelism, even with an unbounded widening degree. For an operation to be compactable, it is required that it not belong to a recurrence. In addition, for memory operations, it is also required that this operation have an access pattern with stride 1.

The replication technique is more versatile than the widening technique: Applying a replication degree of **n** means that the configuration can access **n** independent words in memory or perform **n** independent operations per cycle. Applying the same degree of widening requires a study at compile time to detect *compactable* operations (i.e., **n** single operations that can be unified in one *wide* operation, that can be performed in one *wide* functional unit). The compaction algorithm can be found in [24]. On the other hand, the replication technique has, in general, higher costs than the widening technique:

- Buses: Widening increases the width of the data bus, but not the control and address buses. Besides this, replication increases the number of buses between the register file and the first-level cache, so it requires more ports in the cache memory (multiported caches require more area and have more access time [17]). Also, widening requires no additional address translations, while replication requires several addresses translated per cycle (this affects the number of ports of the TLB, causing an increase in cycle time and die area of the TLB [3]).
- Register file (RF): In our proposal, widening is applied to buses, FPUs, and register file. Every register in the RF increases its width in bits, but they have the same number of ports per bit. Applying replication increases the number of ports per bit. Both techniques increase the RF area and cycle time, but increasing the number of ports per bit has a higher cost than increasing the number of bits per register, as we show in Section 5.
- FPUs: Both techniques require almost the same hardware at the FPU level because they perform the same number of operations per cycle.
- Code size: From the point of view of code generation, widening can reduce the total number of instructions (a single wide operation is the result of compacting multiple operations). This reduction of the code size can reduce the miss rate of the instruction cache and further improve performance.

3.2 Fusion

In order to improve the performance of recurrence-bound loops, the number of cycles needed to perform the operations in the recurrence needs to be reduced. This reduction can be achieved either by reducing the latency of the functional units (which is a technological problem beyond the scope of this paper) or by solving complex operations in the same amount of time. The latter option has been included in the design of some current microprocessors with functional units that execute fused multiply and add (FMA) operations as single instruction:



(a)

(b)

Fig. 5. Graph transformation and impact in performance limits in an architecture with one bus and one FPU: (a) without FMA and (b) with FMA.

$T = (A * C) + B$. 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 several microprocessors, such as the IBM RS/6000 [11], IBM POWER2 [39], MIPS R8000 [10], and MIPS R10000 [42]. In the case of the 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 two cycles to compute an add or a multiply and four cycles to compute an FMA operation providing no latency benefit. The only benefit is the reduction in instruction bandwidth and in the register requirements (no register is required to store the intermediate result). On the contrary, processors like the 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

For example, let us consider the dependence graph shown in Fig. 5a with a recurrence that includes nodes 3, 4, and 5. The recurrence also includes a product followed by an addition. Fig. 5b 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 three to two and reducing the recurrence limit to four cycles. Notice that the number of nodes corresponding to arithmetical operations has also been reduced from four to three.

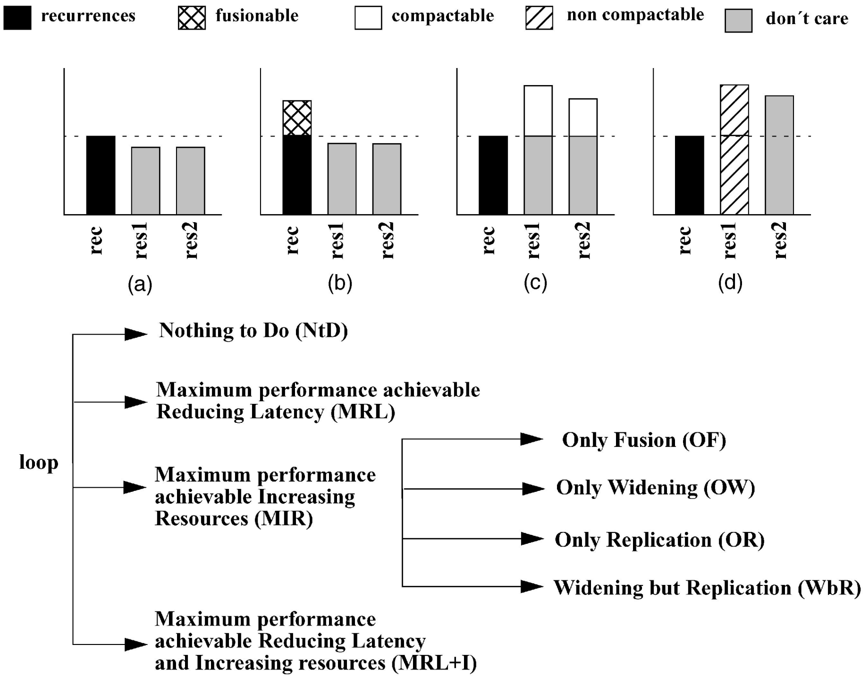
Implementing the FMA functional unit in a microprocessor incurs several costs in terms of area and cycle time. With respect to the FPUs, the area required for the extra hardware needed to implement the FMA operation is practically negligible because the area of a general-purpose floating-point unit is mainly governed by the area of the multiplier [18]. 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 the register cell, which grows as the square of the number of ports (see Section 5). As every FMA functional unit has three read and one write ports, it requires an additional read port to the register file cells.

On the other hand, using FMA functional units can reduce the MII of some loops. It also reduces the need for spill code (because no register is required to store the intermediate result) and reduces the complexity of the scheduled graph, increasing the likelihood of the scheduler finding an optimal schedule [22].

3.3 Classification of Loops

In order to increase the performance of a loop, the technique that affects the most limiting factor of the loop must be used. Several cases can occur: The loop can be recurrence bound without any fusionable operation in the recurrence (Fig. 6a), in which case none of the techniques have any effect on the final performance; if the loop is recurrence bound, but there are fusionable operations in the recurrence, fusion can be applied (Fig. 6b). If the loop is resource bound, the performance can be increased using widening if there are compactable operations (Fig. 6c), fusion if there are fusionable operations or using replication. If there are no compactable operations (Fig. 6d), only fusion (if there are fusionable operations) and replication can be applied. According to that, we can classify the loops in the following way (Fig. 6e):

- NtD *(Nothing to Do)*. These are loops whose performance is limited by recurrences and there are no fusionable operations into the recurrence. The performance of these loops cannot be increased either by using widening, by using replication, or by using fusion.
- MRL (*Maximum performance achievable Reducing Latency).* These loops are recurrence bound loops with fusionable operations into the recurrence that can achieve the maximum possible performance applying only the fusion technique to reduce the latency of the most critical recurrence (i.e., when fusion is applied, the loop becomes a *Nothing to Do* loop).
- MIR (*Maximum performance achievable Increasing Resources*). These are either loops without recurrences or loops with recurrences whose most limiting factor is the resource constraints and there are no fusionable operations into the most limiting recurrence. In all cases, the maximum performance can be achieved if we only apply the replication technique.


recurrences
fusionable
compactable
non compactable
don´t care
rec
res1
res2
loop
Nothing to Do (NtD)
Maximum performance achievable Reducing Latency (MRL)
Maximum performance achievable Increasing Resources (MIR)
Only Fusion (OF)
Only Widening (OW)
Only Replication (OR)
Widening but Replication (WbR)
Maximum performance achievable Reducing Latency and Increasing resources (MRL+I)

(a) (b) (c) (d)

(e)

Fig. 6. (a)-(d) Some types of loops. (e) Classification of loops.

Replication is a very expensive technique and the goal of this paper is to show the potential benefit of combining replication with fusion and widening, so, according to the nature of the operations, MIR loops have been divided in four subtypes:

- OF (*Only Fusion loops*) are loops that can achieve maximum performance applying replication but also applying only the fusion technique (i.e., loops where the most limiting factor is the FPU resources, but, when fusion is applied, the loop becomes *Nothing to Do*).
- OW (*Only Widening loops*) are loops where the limit imposed by the recurrence can be reached by applying only the widening technique. Some of these loops can benefit from applying fusion, but none of them can reach the maximum performance using only fusion.
- OR (*Only Replication loops*) are loops where all the operations that require the most limiting resource are noncompactable (so widening does not affect the final performance). Some of these loops can also benefit from having FMA FPUs, in spite of not achieving the maximum performance.
- WbR (*Widening but Replication loops*) are the ones where there are compactable and noncompactable operations (so some performance benefit can be obtained by widening, but replication is also needed to achieve the maximum performance). Again, some of these loops can benefit from having FMA FPUs, but none of them can reach the maximum performance using only fusion.

- MRL+I (*Maximum performance achievable Reducing Latency and Increasing the resources*). These are loops with recurrences where the maximum performance can be achieved combining replication and fusion, but not using only one. These loops can be divided in subtypes in a similar way as the MIR loops, but there are so few in the workbench (see the next section) that we will not show numbers for the subtypes.

4 THE WORKBENCH AND ITS CHARACTERISTICS

The workbench is composed of 1,180 loops of the Perfect Club [5], which account for 77.82 percent of the total execution time (on an HP 9000/735 workstation). The extended dependence graphs of all the innermost loops have been obtained with the ICTINEO compiler [4]. We have not measured loops with subroutine calls or with conditional exits since they are not suitable for software pipelining. Loops with conditionals in their body have been converted to single basic loops using IF-conversion [2].

Current (and future) advanced compiler transformations can produce final loops with different extended dependence graphs. For instance, techniques such as backward substitution will enhance the performance of loops with recurrences. Also techniques like loop interchange or unroll-and-jam can be used to maximize stride-1 memory accesses, benefiting even more from wide architectures.

Table 1 shows some characteristics of the loops according to the classification of Section 3.3. The results are shown for

TABLE 1
Some Characteristics of the Perfect Club Loops

Program	Nothing to do (NtD)	Maximum performance achievable by								
		Reducing Latency (MRL)	Increasing Resources (MIR)							Red. Lat. + Inc. Res. (MRL+I)
			Only Fusion	Only Widening		Only Replication		Widening but Replic.		
				No FMA	BfFMA	No FMA	BfFMA	No FMA	BfFMA	
ADM	5.3	32.9	0	6.6	27.5	0	5.8	3.1	18.8	0
QCD	44.8	0	0	54.8	0	0.4	0	0	0	0
MDG	1.6	0	0.1	18.9	37.9	0	0	1.7	39.7	0.1
TRACK	21.2	0	0	56.6	13.2	0	3.1	1.3	4.7	0
BDNA	10.3	0.2	0	0.1	2.6	0	0.1	0.1	86.8	0
OCEAN	24.0	6.0	3.5	18.3	3.5	7.1	14.0	0	14.5	9.4
DYFESM	37.0	1.4	0	2.1	58.4	0.1	0.1	0.5	0.4	0
MG3D	0.7	0	0	4.6	19.9	0.3	0.3	2.3	71.9	0
ARC2D	0.4	0	0	6.0	59.0	0.5	16.5	0	17.6	0
FLO52	0	0	0	11.7	79.0	0.1	6.7	0	2.5	0
TRFD	0	0	0	2.9	0	0	0	0.8	96.3	0
SPEC77	26.4	0.3	0.4	3.2	3.9	0.8	20.7	4.4	40.0	0
Total	6.1	1.3	0.3	7.4	22.8	1.0	4.7	1.8	53.7	0.9

an architecture with one bus and two FPUs[1] and the latencies are as follows: A store is served in one 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 four cycles to be executed. The cells in Table 1 show the percentage of cycles spent on each kind of loop. Each row corresponds to a program in the workbench and the last row shows the total cycles spent (notice that it does not correspond to the average because every program spends different amounts of cycles).

The first column shows the cycles spent in loops that cannot benefit from any of the studied techniques (Nothing to Do loops). The second and third column show the percentage of cycles spent in loops which can achieve maximum performance using only the fusion technique, divided in recurrence-bound loops (MRL) and resource-bound-loops (MIR-OF). Columns 3 to 9 show the cycles spent in loops that achieve maximum performance by increasing the number of resources (MIR). These loops are further divided into the ones that can also achieve this maximum using only the fusion technique (OF), the ones that achieve maximum performance using the widening technique alone (OW), the ones in which replication is the only technique that can bring us to the maximum performance (OR), and the ones that can benefit from widening, but require a small degree of replication to achieve the performance limit (WbR). Columns OW, OR, and WbR are further divided in loops that are not affected by FMA FPUs (No FMA) and the ones that can benefit from having FMA FPUs (BfFMA). Finally, the last column corresponds to the MRL+I loops (where replication and fusion are required in order to achieve the maximum performance).

From the results shown in Table 1, the following conclusions can be drawn about the techniques:

- Despite some programs having a high percentage of *Nothing to Do* loops (as QCD or DYFESM), in most of the programs, this percentage is 0 (as FLO52 and TRFD) or very small. On average, 6.1 percent of the cycles are spent in loops that cannot be improved by any of the studied techniques. Even so, having a higher latency or very aggressive configurations can convert this small percentage to critical for the application performance.
- Using FMA can result in an important increase of performance: Despite that, only 1.6 percent of the final cycles are spent in loops that achieve maximum performance using only fusion (columns 2 and 3). Most of these cycles belong to a few programs (like ADM, with 32.9 percent of cycles spent in recurrence-bound loops, cycles that would belong to *Nothing to Do* loops if FMA operations were not available). However, the most important issue regarding FMA is that 82.8 percent of the cycles are spent in loops that can benefit from having this operation (columns 2, 3, 5, 7, and 9), so the potential benefit of this technique is very high.
- Up to 91.4 percent of the cycles are in loops that can achieve maximum performance using only the replication technique, but, as shown in Section 3.1, this is an expensive technique. The widening technique (a cheaper one) can achieve maximum performance in loops that represent 30.2 percent of the total spent cycles (columns 4 and 5), this being a very high percentage in some programs (90.7 percent in FLO52, 69.8 percent in TRACK, 65 percent in ARC2D, 60.5 percent in DYFESM). Also, loops that

---

[1] Preliminary studies show that a relation of two FPUs for each bus is the most balanced configuration. Also, we have based the cost of the computations on the MIPS R10000, which can issue two floating-point and one memory operation per cycle.

TABLE 2
Semiconductor Industry Association (SIA) Predictions in 1997

	2001	2004	2007	2010	2012
λ (μm)	0.18	0.13	0.10	0.07	0.05
Size (mm^2)	385	430	520	620	750
λ^2 per chip (x10^6)	11881	25443	52000	126530	300000
λ^2 / mm^2 (x10^6)	30.86	59.17	100	204.08	400

represent 55.5 percent of the spent cycles can benefit from this technique, even those not achieving the maximum performance (so 85.7 percent of the spent cycles are in loops that benefit from the widening technique).

- Finally, notice that the column with the highest number of cycles (column 9: 53.7 percent) corresponds to loops that, despite requiring replication, benefit from both widening and fusion. Therefore, an architecture combining the three techniques can be very cost-effective.

5 DESIGN CONSIDERATIONS

5.1 Area Cost

To estimate the area cost, we take into account the SIA (*Semiconductor Industry Association*) predictions [37] for the technology size (λ) and the chip size in order to compute the number of λ^2 per chip for the next processor generations (Table 2).

We have estimated the area of a general purpose FPU using the MIPS R10000 processor as a reference. The R10000 processor FPU includes a multiplier, an adder, and a divider. We consider these components as the basic components of a general purpose FPU. With a $0.25\mu\mathrm{m}$ technology, the R10000 FPU requires $12\mathrm{mm}^2$ of area [32]. So, we assume that the area of a FPU is $12\mathrm{mm}^2 \times 16 \times 10^6 \lambda^2/\mathrm{mm}^2 = 192 \times 10^6 \lambda^2$. As the area cost of an FPU is mainly governed by the size of the multiplier [14], we consider the same area for FPUs with and without FMA.

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 percent of the area required by the register cells [18].

To access the register cell of a multiported RF, each port requires one transistor, a select line, and a data line. In addition, a write port requires a second access transistor and a data line. 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. The memory portion of the register cell at a typical dual scalable CMOS register file is a pair of cross-coupled inverters, consisting of four transistors that force a minimum height of 41λ. The memory portion of the cell can accommodate three select lines running widthwise across the cell. Therefore, the height of the cell does not grow until more than three ports are implemented. After that, each port adds 8λ to the eight. The width of a dual ported cell is 50λ. Each additional read port adds 14λ to the width: 8λ for the data line and 6λ for the access transistor. Each additional write port adds 28λ to the width because it requires two data lines and two access transistors [15], [18].

To illustrate the cost difference between the techniques, let's consider several processor configurations labeled XwY. Each one of these configurations has X bidirectional buses (to perform load/store operations) and twice the number of functional units. The width of the resources is Y words. For instance, configuration 4w1 has four buses and eight FPUs and all the resources have a width of one word, while configuration 1w4 has one bus and two FPUs, all of width four words. The latencies are the ones shown at the workbench section. We append F to those configurations in which FMA is implemented.

Table 3 shows the area cost of configurations 4w1, 2w2, 2w2F, and 1w4 for a 64-RF. Each configuration requires two read plus one write port per FPU (except 2w2F, which requires three read and one write ports) and one read plus one write port per bus, so configuration 1w4 (two FPUs and one bus, all of width 4) requires $5R + 3W$ ports. Doubling the replication degree doubles the port requirements. Also, configuration 2w2F has a cost halfway between 4w1 and 2w2 (and better performance than either, see the next section).

5.2 Register File Access Time

A multiported register file follows the scheme shown in Fig. 7. The access time model used in this paper is based on an adaptation proposed [8] for the register file of the CACTI memory model [40]. In the model, the access time of the register file is assumed to be governed by the read time, and can be written as the sum of the following terms:

TABLE 3
RF Area Cost for Several 64-Register File Configurations

Configuration	# buses / width / ports required	# FPUs / width / ports required	Ports	Area of one memory cell (λ^2)	Bits per register	Total RF area (λ^2)
4w1	4 / 1 / 4R+4W	8 / 1 / 16R+8W	20R+12W	170352	64	698x10^6
2w2	2 / 2 / 2R+2W	4 / 2 / 8R+4W	10R+6W	45820	128	375x10^6
2w2F	2 / 2 / 2R+2W	4 / 2 / 12R+4W	14R+6W	65795	128	539x10^6
1w4	1 / 4 / 1R+1W	2 / 4 / 4R+2W	5R+3W	13122	256	215x10^6

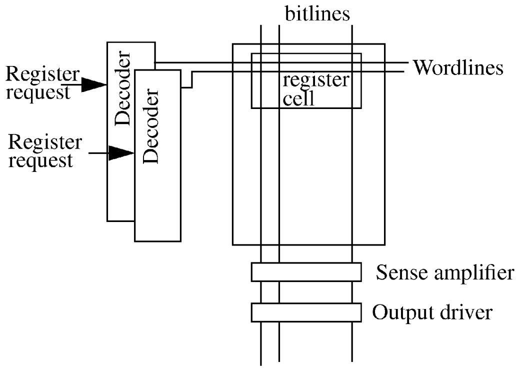


Fig. 7. Multiported register file structure.

- Decoder time: time to decode the register being accessed and select its wordline. This time mainly depends on the number of registers available.
- Wordline time: the time required to drive the select line. It depends on the length of the line (which depends on the size in bits of every register and on the width of every register cell).
- Bitline time: the time delay between the wordline going high and the sense amplifier being able to detect the state of the cell. This delay mainly depends on the height of the cell and on the number of registers.
- Sense time: the time delay through the sense amplifier.
- Outdrive time: the time required to drive the read data onto the internal bus to the ALU.
- Precharge time: the time to precharge the bitlines, comparators and internal decoder bus.

To summarize, the access time is governed by the number of registers, the width of every register, and the size of every register cell (which depends on the number of read and write ports).

Table 4 shows the relative access time for different configurations varying the number of registers (32, 64, 128, and 256), the width of every register (64 bits times the width degree), and the size of the register cell. In order to consider the access time independent of the technology used, all times have been normalized with respect to the time of the 1w1 32-RF configuration.

To reduce the access time, a register file can be partitioned into several RF, maintaining copies of all the data [6]. For example, the RF of a configuration 8w1 can be implemented on a single RF where each cell requires 40 read plus 24 write ports ($8R + 8W$ for the eight buses and $32R + 16W$ for the 16 FPUs). This RF can be also implemented by two identical copies, where all functional units can write in both copies of the RF, but only four buses and eight FPUs read each copy. In this case, $20R + 24W$ ports are required for each copy. Register file partitioning reduces access time at the expense of an increase in RF die area. The configuration 8w1 can be partitioned in one, two, four, or eight blocks, increasing the relative area and decreasing the cycle time, as shown in Fig. 8. Notice that the behavior of the area growth is exponential while the decrease of the access time is logarithmic. A small partitioning, like a 2-partitioning, has a slight increase in area and an important decrease in access time.

6 PERFORMANCE/COST TRADE-OFFS

In this section, we first show an evaluation of the performance considering an "ideal framework." Then, we make some aspects real and analyze the influence of using a heuristic algorithm to do software pipelining and having a bounded register file. Finally, we calculate the costs of the studied configurations and study the performance/cost trade-offs.

6.1 Performance with an Ideal Framework

We have removed some constraints in order to study what performance can be expected from the tested loops. The ideal framework has the following characteristics: We assume that each loop can be scheduled with its *Minimum Initiation Interval*, (i.e., $II = MII$). We consider an unbounded register file, so no spill code is required. Also, the memory is ideal (i.e., all references always hit in the cache).

The processor configurations considered in this section are named *XwY* (i.e., *X* bidirectional buses and twice the number of functional units, all of width *Y*; we append *F* to those configurations in which FMA is implemented, see Section 5.1).

Fig. 9 shows the performance (relative to a baseline configuration: 1w1) achievable using the studied techniques under the described conditions.[2] The configurations shown in the figure are grouped at the horizontal axis by the maximum number of operations that can be performed per cycle (i.e., configurations X_1wY_1 and X_2wY_2 are plotted on the same vertical line if $X_1 * Y_1 = X_2 * Y_2$). For instance, both configurations 1w2 and 2w1 can issue two memory and four arithmetic operations at the same time, but using different techniques. Both configurations are at the label 2 because they can issue twice the number of operations of the baseline configuration (1w1). From these plots, several conclusions can be drawn:

- Configurations based on replication (i.e., configurations *Xw1*, upper dashed plot in Fig. 9) show a progressive performance degradation. This is because aggressive configurations can easily convert resource-bound loops into recurrence-bound loops and these loops cannot benefit by an increase in resources.
- Configurations based on widening (i.e., configurations *1wY*, lower dashed plot in Fig. 9) show even more performance degradation due to noncompactable operations. For instance, in a 1w8 configuration, either eight compactable operations or one noncompactable operation can be issued per cycle; therefore, the presence of noncompactable operations introduces an enormous penalty on these configurations.
- Some of the intermediate configurations (i.e., configurations where replication and widening techniques are combined) also report good performance.

---

[2] Performance for each configuration is estimated by the sum of the cycles required by all loops. The cycles required by a loop is computed as MII times the number of iterations.

TABLE 4
Relative Register File Access Time without Register Partitioning (Baseline: 1w1 32-RF)

Configuration	Register file size							
	FPUs without FMA operation				FPUs with FMA operation			
	32	64	128	256	32	64	128	256
1w1	1	1.05	1.18	1.34	1.08	1.13	1.27	1.43
2w1	1.49	1.54	1.70	1.87	1.66	1.72	1.87	2.06
1w2	1.10	1.15	1.29	1.45	1.20	1.25	1.39	1.56
4w1	2.44	2.51	2.69	2.90	2.78	2.86	3.03	3.26
2w2	1.65	1.72	1.87	2.06	1.85	1.92	2.08	2.27
1w4	1.22	1.27	1.43	1.60	1.33	1.39	1.55	1.72
8w1	4.32	4.41	4.61	4.87	5.00	5.09	5.30	5.57
4w2	2.75	2.82	3.00	3.23	3.13	3.22	3.40	3.64
2w4	1.85	1.92	2.09	2.29	2.08	2.15	2.33	2.53
1w8	1.39	1.45	1.62	1.80	1.52	1.58	1.75	1.95
16w1	8.04	8.15	8.39	8.72	9.38	9.50	9.75	10.09
8w2	4.89	4.99	5.20	5.48	5.67	5.77	5.99	6.28
4w4	3.10	3.18	3.38	3.61	3.54	3.63	3.83	4.08
2w8	2.12	2.20	2.38	2.60	2.38	2.46	2.65	2.87
1w16	1.68	1.75	1.93	2.14	1.84	1.91	2.10	2.32

For example, the behavior of the *2wY* configurations saturates in the same way as the *1wY* configurations, but the saturation point is close to a speed-up of 8 instead of 5. Also, the *Xw2* configurations have performances very close to the *Xw1* configurations.

- FMA functional units can play an important role, especially in aggressive configurations where replication is used (for instance, configuration 1w8F has a performance 2.1 percent better than 1w8, while 8w1F has a performance 9.4 percent better than 8w1).

6.2 Scheduler and Register Constraints

The studied techniques can increase the performance of a loop by reducing its Initiation Interval (II). Regretably, reducing the II can increase the register requirements. If the registers required to schedule a loop on an architecture exceed the number of physical registers, spill code must be inserted in order to free some registers. However, spill code increases the memory traffic and can result in an increase of the II, with the associated performance degradation.

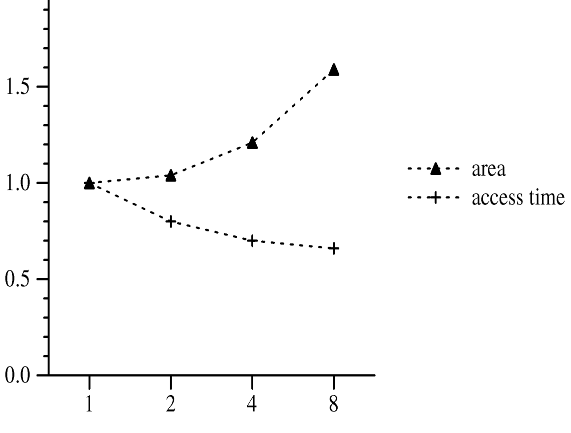


Fig. 8. Behavior of the RF area and cycle time of an 8w1 64-RF configuration with RF partitioning in one, two, four, and eight blocks.

When widening is applied, we also have a wide register file (i.e., in an XwY configuration, all registers are of Y words wide; in our case, 64 bits times Y). For instance, a 32-RF 4w1 configuration has 32 registers of width 1 (i.e., 64 bits), while a 32-RF 4w2 has 32 registers of width 2 (i.e., 128 bits). Notice that if we schedule a loop with compactable operations in a 4w2 configuration, these operations produce two results that are stored in a single wide register, so we have an additional storage capacity; however, if the loop scheduled in the 4w2 configuration has no compactable operations, we do not benefit from this additional capacity.

Fig. 10 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, so it does not require spill code and, therefore, is equivalent to the baseline configuration in Fig. 9. Fig. 10 shows the expected results:[3] 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) improve the performance of all configurations. However, the

---

[3] Notice that the configuration *8w1* does not include the *32-RF* bar; this is because this configuration can produce 24 results per cycle (eight 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.

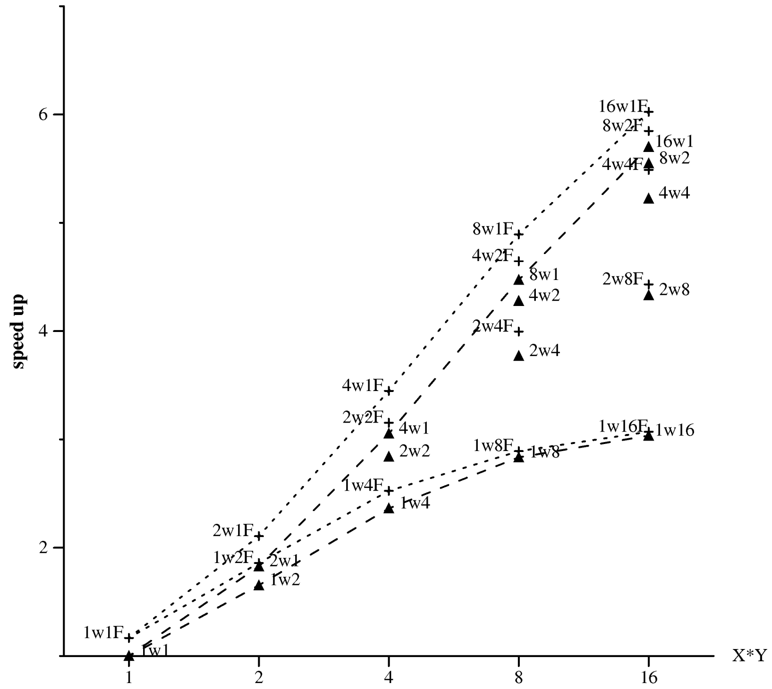


Fig. 9. Speed-up for different configurations XwY and XwYF.

improvement is more noticeable for those configurations with high register pressure.

The results show that, when the configurations become more aggressive, the need for spill code increases, reducing the performance. For example, configuration 4w2 has a performance of 2.25 (with 32-RF), 3.28 (64-RF), 4.39 (128-RF), and 4.76 (256-RF), while the 1w2 configuration achieves almost its maximum performance with a 64-RF.

It is very important to remark that the additional RF capacity of a configuration where the widening technique has been applied reduces the need for spill code. For example, the 8w1 configuration has a theoretical performance greater than the 4w2 configuration, but Fig. 10 shows that the configuration 4w2 with 64-RF has a performance greater than the 8w1 64-RF configuration and the same happens with a 128-RF. Only with a 256-RF does 8w1 have better performance than 4w2. Looking at the results, we can conclude that the additional capacity of the register file, obtained when the widening technique is used, has an important impact on the final performance.

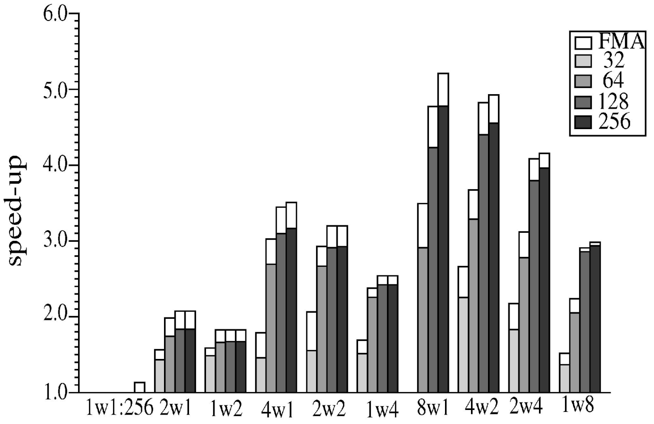


Fig. 10. Performance of some processor configurations for several sizes of the register file. Baseline: configuration 1w1 with a 256-RF.

Another important point can be observed in Fig. 10: There is a significant difference between the performance reported in this section and the theoretical performance shown in Section 6.1. For instance, there is an increase of the theoretical performance of 7.5 percent from configuration 8w1 to configuration 8w1F. However, when the loop is scheduled with a 64-register file, this difference grows up to 20.5 percent. This difference is due to several factors that have an important effect on the final performance.

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.
- Cycles due to the scheduler heuristic: These are the cycles added by the scheduler when it fails to find an optimal schedule.

Each bar in Fig. 11 shows the distribution of cycles contributed by each one of these components for all the configurations with $X * Y = 2$ and $X * Y = 4$. Other configurations have been omitted in order to simplify the figure. The plot is divided in blocks of three columns representing processor configurations with and without FMA FPUs; the 3 columns of each block represent cycles for register file sizes of 32, 64, and 128-registers (from left to right). For each column, the total amount of cycles (vertical axis) is divided in the components previously described:

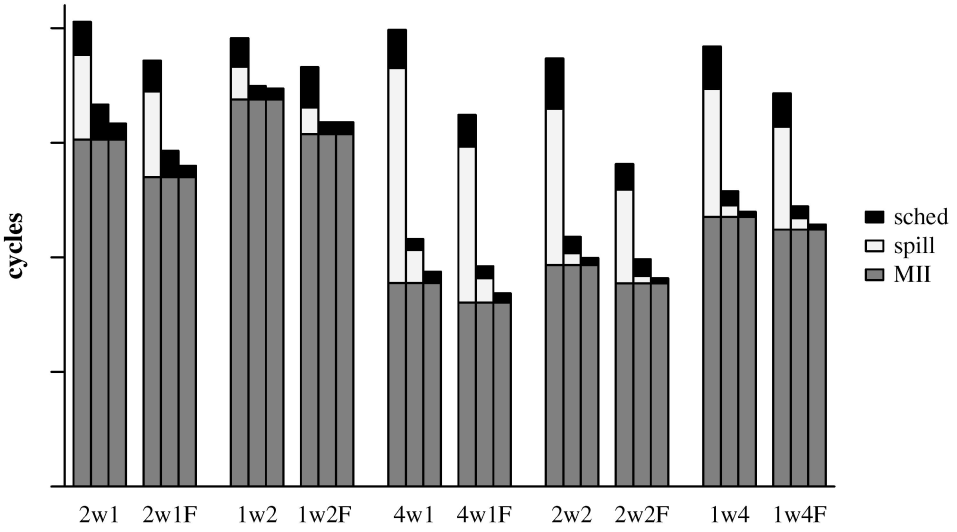


Fig. 11. Distribution of the spent cycles for configurations with factors $X * Y = 2$ and 4. For each configuration, the three columns represent (from left to right) register files of 32, 64, and 128-registers.

The dark gray part represents the minimum theoretical cycles, so it is independent of the register file size. The light gray part shows the spill code cycles and the black part shows the cycles added by the scheduler. The results for a 256-RF are practically indistinguishable from the 128-RF plot, so they are not shown.

From the analysis of Fig. 11, we can conclude that, when the scheduling heuristic and the spill code are taken into account, mixing the widening and the replication techniques and using fusion have an impact on the final performance greater than what we can expect if we only take into account the theoretical analysis.

- With a small RF, some aggressive configurations have an enormous penalty in terms of spill code. For instance, configuration 4w1 with a 32-RF spends more cycles due to spill code than due to the MII. With this RF size, configurations 2w2 and 1w4 have better performance than 4w1 (6.7 percent and 3.8 percent, respectively), whereas the latter has the best theoretical performance (8.7 percent better with respect to 2w2 and 32.5 percent with respect to 1w4).
- The use of FMA functional units reduces the MII, but it also reduces the cycles due to spill code and the scheduler. The cycles of spill code are reduced because it does not require a register to store the intermediate result (e.g., the number of cycles due to spill code are reduced by 37 percent when FMA is used in configuration 4w1 with a 32-RF). The cycles due to the scheduler are reduced for two reasons: the reduction of operations in the loop and the reduction of the spill code required. This makes the graph less complex and the scheduler has more opportunities to find a schedule closer to the optimal (e.g., the number of cycles added by the scheduler is reduced by 20.9 percent when FMA is used in configuration 4w1 with a 32-RF).
- Finally, the cost of configurations affects the decision of the architecture to be chosen. For instance, configuration 2w2 with a 128-RF has a performance 8.3 percent better than 4w1 with a 64-RF; if the first one has also a smaller cost, this will be the one to choose. In the next section, we will study the cost of the evaluated techniques.

6.3 Area and Cycle Time Constraints

In this section, we perform a study of the performance-cost trade-offs. We first analyze the individual effects of some parameters. Then, we show the performance/cost ratio for the most significative configurations (of a total of 360 configurations evaluated). Finally, we show the best configurations for the technology generations considered.

Configurations are labeled XwY(Z:n) (i.e., X buses and 2*X FPUs, all of width Y, with an RF of Z registers of width Y, partitioned in n-blocks). We append *F* to those configurations in which FMA is implemented.

Assuming the SIA predictions and our area cost models, Fig. 12 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). From the analysis of the layout of some current microprocessors, we consider that 20 percent of the area devoted to the FPU core is a good limit (most of current processors are well below this limit), so each horizontal line represents this limit for each technology generation and, therefore, defines the set of implementable configurations for this technology.

For each implementable configuration, we compute its cycle time, assuming that the cycle time is determined by the RF cycle time. We adapt the latency in cycles of the FPUs to match this cycle time and we perform the scheduling to find the cycles required to execute our benchmark set.[4] Each FPU requires an amount of time to perform one operation; its latency in cycles depends on the processor cycle time. For instance, let's assume an FP adder

---

[4] The cycles required are calculated in the same way as those in Section 6.1, but, in this case, the cycles per iteration are the Initiation Interval once the graph has been scheduled (with spill code if necessary). The cycles required to execute all the loops times the cycle time give us the final performance.

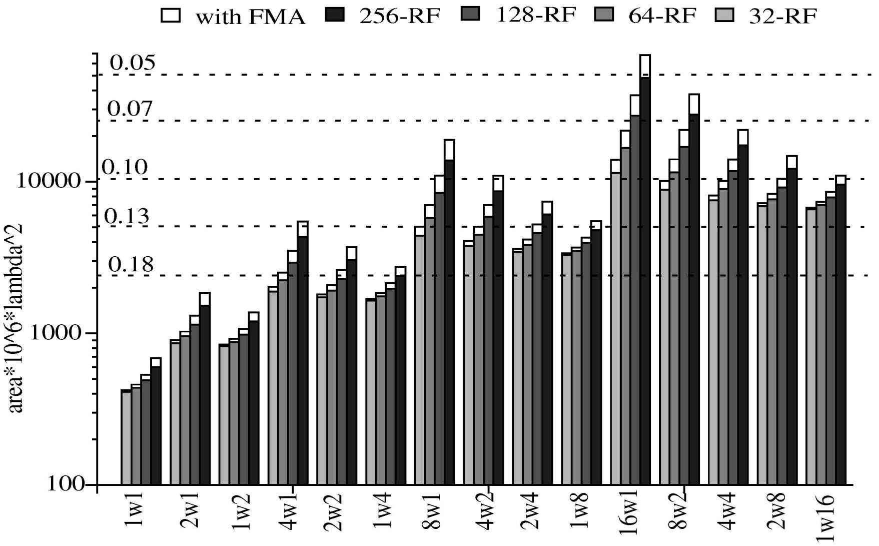


Fig. 12. Implementable configurations for each technology generation considered.

that requires N nanoseconds to perform an addition, corresponding to a latency of four cycles. If the RF access time forces us to double the cycle time, an addition still requires N nanoseconds, so the latency in cycles of the addition is now two. This adaptation is important because the latency of each operation determines the schedules of the loops. For this reason, we compare configurations modifying the latency of the FPUs in order to make them coherent with the processor cycle time. The four models we have tested are listed in Table 5.

We assume the 4-cycles model for configuration 1w1. Each configuration considered can be classified into a cycle model depending on its relative (from the 1w1 configuration) cycle time. A configuration with a relative cycle time T_c belongs to the c-cycles model, where $c = \lceil 4/T_c \rceil$. For example, the 2w4(32:1) configuration has a relative cycle time of 1.85 (i.e., 3-cycles model), while the 2w4(128:1) configuration has a relative cycle time of 2.09 (i.e., 2-cycles model), and the 2w4(128:2) configuration has a relative cycle time of 1.80 (i.e., 3-cycles model).

Before going into the final results, let us analyze the individual effects of some parameters on the configurations evaluated (Fig. 13):

- Number of registers: Having large register files reduces the register pressure and the need for spill code. However, the increase in the cycle time may counteract this gain. For example, Fig. 13a shows the performance/cost ratio 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 register file does not reduce the execution 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 percent of the total chip area) or suffer a decrease in performance because a small increase in IPC (instructions per cycle) is counteracted by a high increase of the cycle time. For instance, Fig. 13b 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 reports higher ILP returns, cycle time puts configurations based on small degrees of widening in a better position, as shown in Fig. 13c for configurations where $X * Y = 8$.
- Fused Multiply-Add: FMA returns good performance relative to its low implementation cost. The three plots in Fig. 13 also show the performance for the same configurations when FMA is included (the cross marks). For instance, configuration *8w1F(128:8)* performs 21.1 percent better than *8w1(128:8)*, with an increment of only 8.3 percent in the area. Also, configuration *2w4F(128:2)* has a performance 2 percent better than configuration *4w2(128:4)* and only has 72 percent of its area requirements.

TABLE 5
Cycles/Operation for the Cycle Models Tested

cycle model	cycles of operations			
	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

Operations div and sqrt are not pipelined; other operations are fully pipelined.

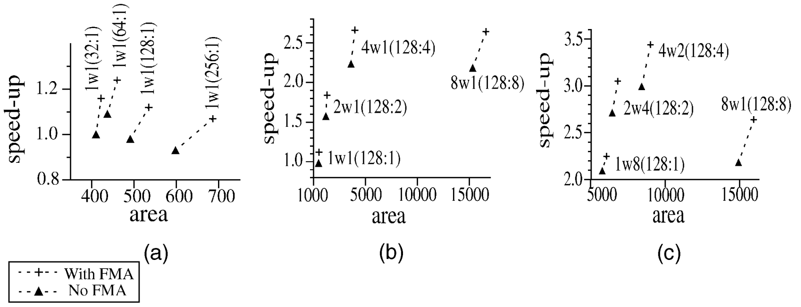


Fig. 13. Effect of (a) increasing the RF size, (b) replication, and (c) different ways of implementing a configuration with the same peak performance. All figures compare FPUs with and without FMA.

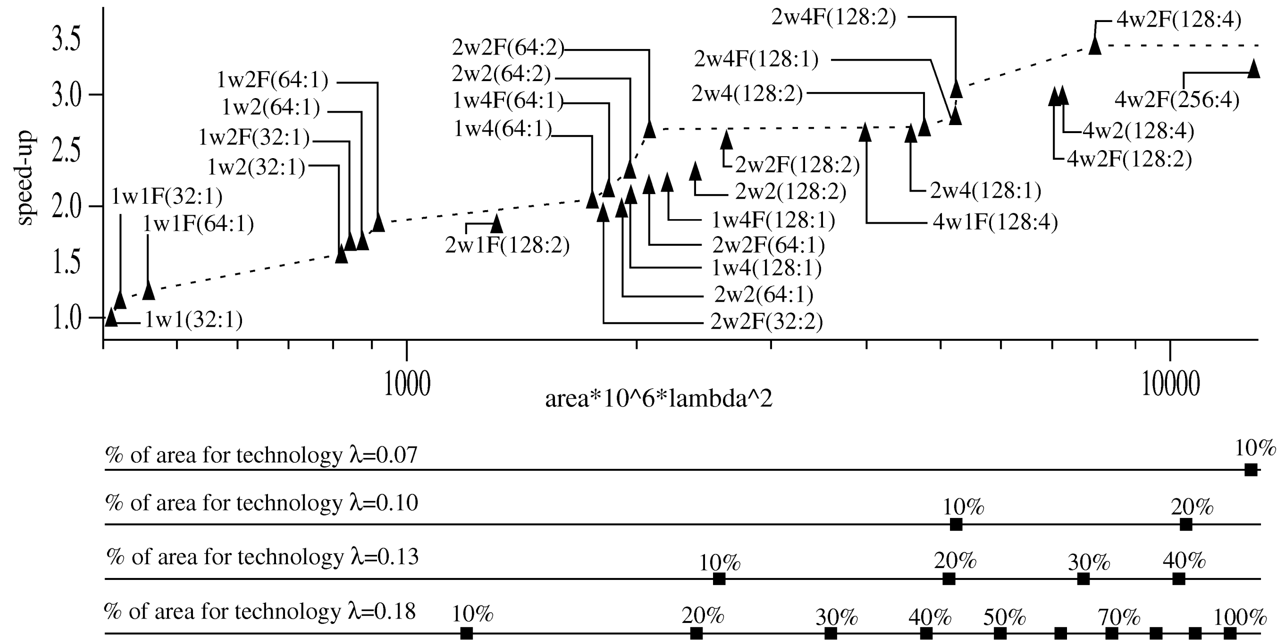


Fig. 14. Speed up and area cost of several configurations. The dashed line links the configurations with the best performance for their cost ("eligible" configurations). The lines below the figure show, for each technology, the percentage of area that the horizontal axis represents (it has logarithmic scale).

We have tested a total of 360 configurations. Fig. 14 shows the speed-up and area cost for some of them (the ones with the best performance/cost ratio). The dashed line links the "eligible" configurations. A configuration is "eligible" if there is no configuration that can achieve the same performance, or better, with a smaller cost. Notice that none of the most aggressive configurations (like 8w1 or 16w1) are in this figure due to their high cost. The configurations that offer better performance are the ones that combine small degrees of replication and widening. Notice that, usually, configurations with FMA FPUs offer a considerable speed-up with a small increase in area cost. At the bottom of this figure, the reader can find the percentage of area devoted to the FPU core for each technology. Technology $\lambda = 0.05$ is not shown since the best configurations are the same as that of $\lambda = 0.07$ and the only difference is the relative percentage of area required. Notice that the area (horizontal axis) has logarithmic scale.

In order to complement the results of Fig. 14, Fig. 15 shows, for the previous technologies, the five configurations that achieve the best performance. For each technology, we have also highlighted the "eligible" configurations among the top five (black triangles). Notice that all except two "eligible" configurations implement the FMA operation in their FPUs. For example, for a technology of $\lambda = 0.13$ (Fig. 15b), the configuration with best performance is *2w4(128:2)*, using 18.7 percent of the total chip area. The configuration with the second best performance is *2w2F(64:2)*, which achieves 99.3 percent of the performance of the first one using only 8.18 percent of the total chip area. Configuration *2w4F(128:2)* is not included in the plot because it requires 20.6 percent of the total chip area (more than 20 percent) but offers a performance 12.5 percent greater than 2w4(128:2).

We can conclude that combining small degrees of replication and widening results in better performance than using only replication when we have a technological limit. Also, we can conclude that using FPUs that implement the FMA operation has some costs, but the benefits that it offers overcome these costs.

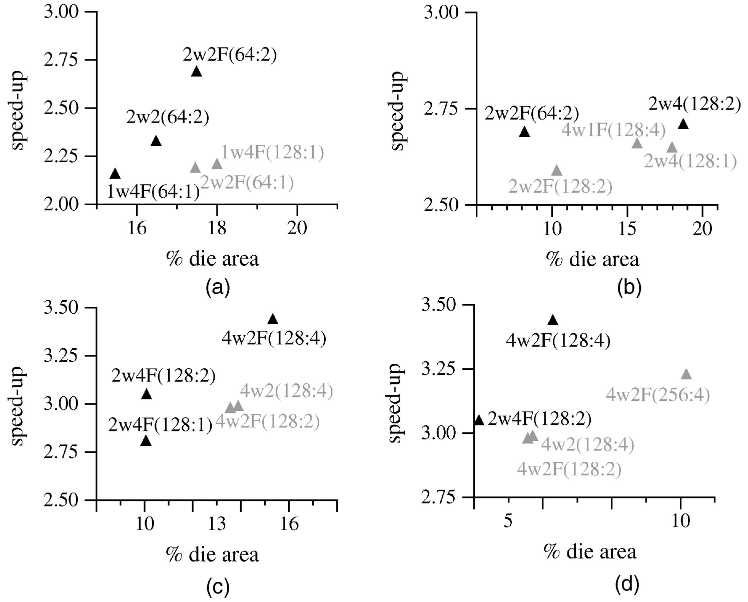


Fig. 15. Top five configurations for technology (a) 0.18, (b) 0.13, (c) 0.10, and (d) 0.07. For technology 0.05, the results are not shown because the top five configurations are the same. In all cases, the possible increment of the clock speed has not been taken into account.

7 CONCLUSIONS

In order to exploit the ILP available in numerical applications, aggressive processor configurations capable of executing a high number of operations per cycle 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). Also, fused multiply-add functional units (fusion technique) increase the number of operations performed by cycle and improve performance in recurrence-bound loops that contain multiply-add chains in their critical recurrence.

We have proposed combining replication and widening in the design of VLIW processors oriented to execute numerical applications. We applied widening to the floating-point functional units, the register file, and the buses between the register file and the first-level data cache. We have presented a study of the ILP with no register constraints from which we can conclude that applying only the replication technique offers the best theoretical performance. However, when a limited register file is used, the increase of storage capacity due to wider registers can reduce the need for spill code. The results show that this additional capacity has an important impact on the final performance (e.g., with a 128-RF, configuration 4w2 achieves better performance than configuration 8w1, whereas the latter has the best theoretical performance).

We have also analyzed the effects of FMA on resource-bound loops and on recurrence-bound loops. With no register constraints, FMA units provide a significant advantage. This is because FMA units increase the peak number of 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 and the influence of the scheduler. The use of FMA units reduces the register requirements, reducing the degradation due to spill code. In addition, the compiler has a simpler task since there are fewer operations to schedule (there are fewer spill load/stores and some pairs of multiply-adds have been collapsed into a single operation), obtaining better schedules.

We have estimated the cost of a large number of configurations that combine the three techniques. We compare the performance of the configurations that can be built for the next processor generations (according to the SIA predictions). The performance has been calculated assuming that the register file access time limits the processor cycle time. From this study, we conclude that, for a given technology, the best performance is obtained when combining a small degree of replication and widening in the hardware resources, using FMA functional units. For instance, in configuration 4w1(128:4), only replication has been applied, while configuration 2w2(128:2) can issue the same number of operations per cycle by combining widening and replication. Both configurations can be implemented using a technology of 0.10λ. Configuration 2w2(128:2) has a performance 3 percent better than 4w1(128:4). Also, the area cost of 2w2(128:2) is only 64 percent of the area cost of 4w1(128:4). If we apply fusion to configuration 2w2(128:2), the area cost grows 11 percent, but the performance increases 12 percent. Comparing 2w2F(128:2) with 4w1(128:4), the first one has a performance 16 percent better having an area requirement of 71 percent of the second configuration.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Culture and Education of Spain under contract TIC 98-0511 and by CEPBA (European Centre for Parallelism of Barcelona). The authors wish to thank the anonymous referees for their valuable suggestions.

REFERENCES

[1] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan, "Software Pipelining," *ACM Computing Surveys,* vol. 27, no. 3, pp. 367-432, Sept. 1995.

[2] J.R. Allen, K. Kennedy, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Proc. 10th. Symp. Principles of Programming Languages,* Jan. 1983.

[3] T.M. Austin and G.S. Sohi, "High-Bandwidth Address Translation for Multiple-Issue Processors," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA-23),* pp. 158-167, May 1996.

[4] 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," Technical Report UPC-DAC-1996-61, Technical Univ. of Catalunya, Dec. 1996.

[5] M. Berry, D. Chen, P. Koss, and D. Kuck, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," Technical Report 827, CSRD, Univ. of Illinois at Urbana-Champaign, Nov. 1988.

[6] A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs," *Proc. 25th Int'l Symp. Microarchitecture (MICRO-25),* pp. 292-300, Dec. 1992.

[7] J.C. Dehnert and R.A. Towle, "Compiling for Cydra 5," *J. Supercomputing,* vol. 7 nos. 1/2, pp. 181-227, May 1993.

[8] K.I. Farkas, "Memory-System Design Considerations for Dynamically-Scheduled Microprocessors," PhD dissertation, Univ. of Toronto, 1997.

[9] J.A. Fisher, P. Faraboschi, and G. Desoli, "Custom-Fit Processors: Letting Applications Define Architectures," *Proc. 29th Int'l Symp. Microarchitecture (MICRO-29),* pp. 324-335, Dec. 1996.

[10] P.Y.T. Hsu, "Design of the FTP Microprocessor," *IEEE Micro,* vol. 14, no. 2, pp. 23-33, Apr. 1994.

[11] IBM, Special Issue on the RS/6000, *IBM J. Research and Development,* vol. 34 no. 1, Jan. 1990.

[12] INTEL, "Pentium III Processor: Developer's Manual,"Intel Technical Report available at http://developer.intel.com/design/PentiumIII, 1999.

[13] J. Janssen and H. Corporaal, "Partitioned Register File for TTA," *Proc. 28th Int'l Symp. Microarchitecture (MICRO-28),* pp. 303-312, Nov./Dec. 1995.

[14] R.M. Jessani and M. Putrino, "Comparison of Single- and Double-Pass Multiply-Add Fused Floating-Point Units," *IEEE Trans. Computers,* vol. 47, no. 9, pp. 927-937, Sept. 1998.

[15] R. Jolly, "A 9-ns 1.4 Gigabyte 17-Ported CMOS Register File," *IEEE J. Solid-State Circuits,* vol. 25, no. 10, pp. 1407-1412, Oct. 1991.

[16] R.B. Jones and V.H. Allan, "Software Pipelining: A Comparison and Improvement," *Proc. 23rd Int'l Symp. Microarchitecture (MICRO-23),* pp. 46-46, Nov. 1990.

[17] T. Juan, J.J. Navarro, and O. Temam, "Data Caches for Superscalar Processors," *Proc. 11th. Int'l Conf. Supercomputing (ICS-11),* pp. 60-67, July 1997.

[18] C.G. Lee, "Code Optimizers and Register Organizations for Vector Architectures," PhD dissertation, Univ. of California at Berkeley, May 1992.

[19] J. Llosa, E. Ayguadé, and M. Valero, "Quantitative Evaluation of Register Pressure on Software Pipelined Loops," *Int'l J. Parallel Programming,* vol. 26, no. 2, pp. 121-142, 1998.

[20] J. Llosa, M. Valero, and E. Ayguadé, "Heuristics for Register-Constrained Software Pipelining," *Proc. 29th Int'l Symp. Microarchitecture (MICRO-29),* pp. 250-261, Dec. 1996.

[21] J. Llosa, M. Valero, E. Ayguadé, and A. González, "Modulo Scheduling with Reduced Register Pressure," *IEEE Trans. Computers,* vol. 47, no. 6, pp. 625-638, June 1998.

[22] D. López, J. Llosa, E. Ayguadé, and M. Valero, "Impact on Performance of Fused Multiply-Add Units in Aggressive VLIW Architectures," *Proc. 1999 Int'l Conf. Parallel Processing (ICPP-99),* pp. 22-29, Sept. 1999.

[23] D. López, J. Llosa, M. Valero, and E. Ayguadé, "Widening Resources: A Cost-Effective Technique for Aggressive ILP Architectures," *Proc. 31st Int'l Symp. Microarchitecture (MICRO-31),* pp. 237-246, Nov.-Dec. 1998.

[24] D. López, M. Valero, J. Llosa, and E. Ayguadé, "Increasing Memory Bandwidth with Wide Buses: Compiler, Hardware and Performance Trade-Off," *Proc. 11th Int'l Conf. Supercomputing (ICS-11),* pp. 12-19, July 1997.

[25] "Intel HP Make EPIC Disclosure," *Microprocessor Report,* vol. 11, no. 14, Oct. 1997.

[26] "AltiVec Vectorizes PowerPC," *Microprocessor Report,* vol. 12, no. 6, May 1998.

[27] "TI Aims for Floating-Point DSP Lead," *Microprocessor Report,* vol. 12, no. 12, Sept. 1998.

[28] "MAP1000 Unfolds at Equator," *Microprocessor Report,* vol. 12, no. 16, Dec. 1998.

[29] "MAJC Gives VLIW a New Twist," *Microprocessor Report,* vol. 13, no. 12, Sept. 1999.

[30] "Merced Shows Innovative Design," *Microprocessor Report,* vol. 13, no. 13, Oct. 1999.

[31] "Sun Makes MAJC with Mirrors," *Microprocessor Report,* vol. 13, no. 14, Oct. 1999.

[32] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* pp. 2-11, Oct. 1996.

[33] B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Int'l Symp. Microarchitecture (MICRO-27),* pp. 63-74, Nov. 1994.

[34] B.R. Rau and J.A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," *J. Supercomputing,* vol. 7, nos. 1/2, pp. 9-50, May 1993.

[35] B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proc. 14th Ann. Microprogramming Workshop,* pp. 183-197, Oct. 1981.

[36] B.R. Rau, M. Lee, P. Tirumalai, and P. Schlansker, "Register Allocation for Software Pipelined Loops," *Proc. SIGPLAN '92 Conf. Programming Language Design and Implementation (PLDI-92),* pp. 283-299, June 1992.

[37] Semiconductor Industry Assoc., "The National Technology Roadmap for Semiconductors,"San Jose, Calif., 1997.

[38] T. Watanabe, "The NEC SX-3 Supercomputer System," *Proc. CompCon91,* pp. 303-308, 1991.

[39] S.W. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM J. Research and Development,* vol. 38, no. 5, pp. 493-502, Sept. 1994.

[40] S.J.E. Wilton and N.P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE J. Solid-State Circuits,* vol. 31, no. 5, pp. 677-688, May 1996. See also: http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[41] M. Wolfe, *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[42] K.C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16 no. 2, pp. 28-40, Mar. 1996.

David López received the degree in computer science in 1991 and the PhD degree in computer science in 1998, both from the Polytechnic University of Catalonia (UPC), Barcelona, Spain. In 1991, he joined the Computer Architecture Department at UPC, where he has been lecturing in computer organization and where he is currently an associate professor. His research interests include computer graphics, microprocessor architecture, and compilation techniques.

Josep Llosa received the degree in computer science in 1990 and the PhD degree in computer science in 1996, both from the Polytechnic University of Catalonia (UPC), Barcelona, Spain. In 1990, he joined the Computer Architecture Department at UPC, where he is currently an associate professor. His research interests include processor microarchitecture, memory hierarchy, and compilation techniques with a special emphasis on instruction scheduling.

Mateo Valero obtained the telecommunication engineering degree from the Polytechnic University of Madrid in 1974 and the PhD degree from the Polytechnic University of Catalonia (UPC), Barcelona, Spain, in 1980. He is a professor in the Computer Architecture Department at UPC. His current research interests are in the field of high performance architectures, with special interest in the following topics: processor organization, memory hierarchy, interconnection networks, compilation techniques, and computer benchmarking. He has published approximately 200 papers on these topics. He served as the general chair for several conferences, including ISCA-98 and ICS-95, and has been an associate editor for the *IEEE Transactions on Parallel and Distributed Systems* for three years. He is a member of the subcommittee for the Ecker-Mauchly Award. Dr. Valero has been honored with several awards, including the Narcis Monturiol, presented by the Catalan Government, the Salvà i Campillo presented by the Telecommunications Engineer Association and ACM, and the King Jaime I by the Generalitat Valenciana. He is the director of the C4 (Catalan Center for Computation and Communications). Since 1994, he has been a member of the Spanish Engineering Academy and, since January 2001, he has been a fellow of the IEEE.

Eduard Ayguadé received the engineering degree in telecommunications in 1986 and the PhD degree in computer science in 1989, both from the Universitat Politècnica de Catalunya (UPC), Spain. Since 1987, he has been lecturing on computer organization and architecture and optimizing compilers. Currently, and since 1997, he is a full professor in the Computer Architecture Department at UPC. His research interests cover the areas of processor microarchitecture and memory hierarchy, parallelizing compilers for high-performance multiprocessor systems, and tools for performance analysis and visualization. He has published more than 100 papers on these topics and participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs.