12th International Conference on Supercomputing (ICS-12), pp 441-448, Melbourne, Australia. July 13-17, 1998.

Resource Widening Versus Replication: Limits and Performance-Cost Trade-off

David López, Josep Llosa, Mateo Valero and Eduard Ayguadé

{ david | josepll | mateo | eduard }@ac.upc.es

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
Campus Nord, Mòdul D6, Jordi Girona 1-3, 08034 Barcelona, SPAIN

Abstract

A balanced increase of memory bandwidth and computational capabilities is going to be one of the trends in the design of near future high-performance microprocessors. Alternative solutions are foreseen for the organization of their resources, mainly based on different degrees of resource replication and/or adaptation of resources to the most frequently found operations in highly performance demanding applications. For instance, doubling the width of buses between the register file and the first-level data cache is an example of design that attains similar performance results than doubling the number of buses in numerical applications. In this paper we evaluate the cost/performance trade-off of a wide set of design alternatives oriented towards having high memory bandwidth and computing capabilities in future architectures. Different degrees of resource replication and widening are the basis for the generation of the design space. Performance evaluation is based on the results obtained for a large number of inner loops in the Perfect Club benchmarks. Implementation costs for the register file and functional units are estimated for different foreseen integration technologies which allow us to analyse their future availability. The results show that replicating is most effective in terms performance but results in an unafordable cost while widening has a much smaller cost but less performance. Combining a small degree of widening and replication results in the best performance/cost ratio.

1. Introduction

Current high-performance microprocessors are based upon hardware techniques that boost the potential instruction-level parallelism they can offer. These microprocessor architectures basically make use of deeper pipelines that reduce the cycle time and wider instruction issue units that allows the simultaneous execution of several instructions per cycle. As the number of transistors on a single chip continues to grow, it is important to think about processor organizations that make an efficient use of them and provide the memory bandwidth and computing capabilities needed by current performance demanding applications, specially those based on floating point computations.

A balanced increase of computational capabilities and memory bandwidth is one of the current trends in the design of these high-performance microprocessors. The increase is either attained by *replicating* resources (for instance, doubling the number of buses or functional units) or by making them more complex. Making resources more complex can be achieved either by *fussing* several dependent operations [12] with a reduced latency (for instance a floating point multiply followed by a dependent add) or by *widening* the resources (for instance doubling the width of buses between the register file and the first-level data cache instead of doubling the number of buses).

Having such a potential instruction-level parallelism (ILP) in microprocessors has driven hardware designers and compiler writers to investigate aggressive techniques for exploiting program parallelism at the lowest level. Software pipelining [8] is a compilation technique to extract ILP from innermost loops by overlapping the execution of consecutive loop iterations. In a software pipelined loop, the *Initiation Interval* (*II*) is the number of cycles between the initiation of successive iterations [14]. The Initiation Interval between two successive iterations is bounded either by recurrences in the dependence graph (*RecMII*) or by resource constrains of the architecture (*ResMII*) [4, 15]. This lower bound is termed the *Minimum Initiation Interval* (*MII*) where *MII = max (ResMII, RecMII)*, so the reduction of the greatest of these two values is required to increase performance.

Reducing the *ResMII* bound can be achieved by increasing the number of operations done by cycle, both in computation and memory access instructions. In computation instructions, the increment of operations done by cycle can be overtaken by increasing the number of functional units and/or by solving complex operations in the functional unit. In memory access instructions, low latency and high bandwidth memory subsystems are required. Low latency memory accesses have been traditionally achieved by using cache memories, which have also contributed to increase the bandwidth of the memory subsystem. However, heavily exploiting ILP implies high memory bandwidth demands (particularly in numeric applications). To meet the high bandwidth requirements, one can increase the number of buses and/or increase their width moving several (consecutive) words per access (use of *wide buses*).

Having complex resources complicates the compilation process; several code reorganization and compaction techniques have to be applied to detect sequences of operations that benefit from having these resources. For example, the same memory access that occur in different iterations of a software pipelined loop would benefit from having wide buses if consecutive accesses have stride one; although vectorizing techniques could be used to detect wide accesses, [11] presents a compilation technique that compacts independent memory operations that access consecutive locations, as well as consecutive accesses to vectors. Loop unrolling, which has been widely used to match the resource requirements of a loop body and the resources available in the architecture [6], is the basic transformation used to find compactable operations across loop iterations.

This paper focuses on the performance evaluation of different processor configurations based on the application of different degrees of replication and widening, both applied to functional units and to memory access ports. For the evaluations, we use 1180 loops that account for 78% of the execution time of the Perfect Club [2], and a modulo scheduler module that applies the appropriate unrolling of loop iterations and compaction of operations before estimating the *II* for each loop with a processor configuration. The performance of each configuration is traded-off with the register file and functional units chip area implementation cost. We will conclude that both techniques are complementary: applying a relatively small degree of widening and replication results in a better performance/cost ratio than simply applying replication.

The paper is organized as follows: Section 2 outlines the benefits of using replication and/or widening and the implications at the different levels of the system, and illustrates, with an example, the performance limits of applying both techniques; it also includes a classification of the loops according to the different performance limiting factors. Section 3 evaluates the performance of a set of processor configurations where different degrees of replication and/or widening are used. Section 4 projects the configurations evaluated in Section 3 into the current and future technological trends, and Section 5 trades-off both aspects to conclude the most interesting cost/performance configurations for the near future.

2. Replication and widening: a closer look

In this section we describe the two techniques that are evaluated in this paper: replication and widening, applied to both functional units and memory access ports. We first analyze the implications of both techniques at the different levels of the architecture. Then we illustrate the factors that may limit the performance obtained out of a software pipelined loop and how these two techniques affect its performance.

2.1 The techniques and their implications

In order to increase the number of operations performed by cycle, the resources of the processor must be increased. In this paper we analyze the trade-off between two options: resource *replication* and *widening*. Replication consists on increasing the number of resources by adding more independent functional units; resource widening consists on increasing the number of operations that a single functional unit can perform per cycle (i.e. functional units that oparate with short vectors).

Figure 1 shows the use of both techniques to access the memory system. Figure 1.a shows a basic processor configuration with a single bidirectional bus, which can perform one memory access per cycle. A higher memory bandwidth can be attained by adding another bus (*replication*) having a maximum of two memory accesses per cycle (Figure 1.b). An alternative is to duplicate the width of the bus (*widening*) as shown in Figure 1.c so that two consecutive memory positions are accessed per cycle. A wide bus requires the two accesses to be independent (i.e. there are no dependences between them) and the two accesses have to be to consecutive memory positions (i.e stride 1). Therefore replication is more versatile and can result in bigger speed-ups than widening, as we show in Section 3. However, bus replication has several costs and drawbacks at different levels of the memory hierarchy that make widening an attractive approach. Both techniques can be used together (Figure 1.d) with a better performance/cost ratio (Section 6). Next we show some of the costs and drawbacks of increasing the number of buses.

At the processor level, doubling the number of buses requires doubling the number of load/store units. Also, doubling the number of load/store units requires additional register-file ports for each additional load/store unit and 1 write port and 1 read port in each register file (integer and floating) for loading/storing the data. Unfortunately, the access time of a multiported device, such as the register file, increases with the number of access ports [18]. Moreover, there is a noticeable area penalty (added to the extra area required for the additional load/store unit), since in CMOS, the area of a multiported device is proportional to the square of the number of ports [7, 9].

In order to allow multiple memory accesses per cycle a multi-

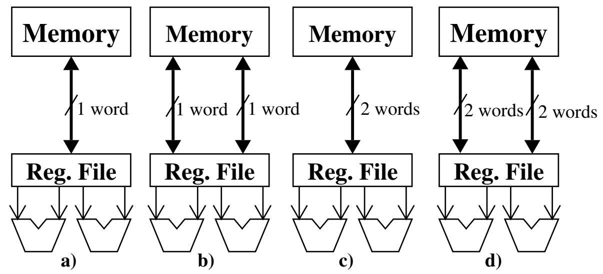


Figure 1 : Different bus configurations: a) 1 single bus, b) 2 single buses (replication), c) 1 wide bus (widening), and d) 2 wide buses (replication and widening).

ported cache is required. Increasing the number of ports of a cache has the same drawbacks than register files. To implement a dual-ported cache, the Alpha 21164 [5] duplicates the data cache and maintains identical copies of the data in each cache. This implementation doubles the die area of the primary data cache. Another option to implement a dual-ported data cache, is to split the data cache into two (or more) independent banks [17]. However, bank conflicts can reduce the effective bandwidth. Besides, it has additional die area cost due to the crossbar between the load/store units and the banks, which can also increase cache access time.

If address translation has to be performed before (physically indexed, physically tagged data cache) or while (virtually indexed, physically tagged data cache) accessing the data cache, multiple translations must be performed per cycle. If this is the case, the TLB might be in the critical path of the processor. Multiporting the TLB (as in the case of the register file) can increase cycle time as well as require some extra die area. Some processor designs implement multi-level TLBs [3]. In this way the first level can be a small multiported TLB providing multiple translations per cycle and the second level can be a bigger single ported TLB providing a higher hit rate. However, a single ported TLB will provide the same, or even better, hit rate, requiring less area.

Resource widening can also be applied to the functional units. Figure 2.a shows a basic processor with a single functional unit. In order to have more computational capabilities, we can add another functional unit (replication) as shown in Figure 2.b. This duplicates the number of ports to access the register file with the aforementioned area and access time costs. An alternative is to duplicate the width of the functional unit as well as the width of the registers and register file ports (widening) as shown in Figure 2.c. Also, as in the case of wide buses, replication is more versatile and may result in bigger speed-ups than widening.

The main drawback of resource replication is the grow in number of access ports to the register file which increases the its access time and area. Although widening also increases the area of

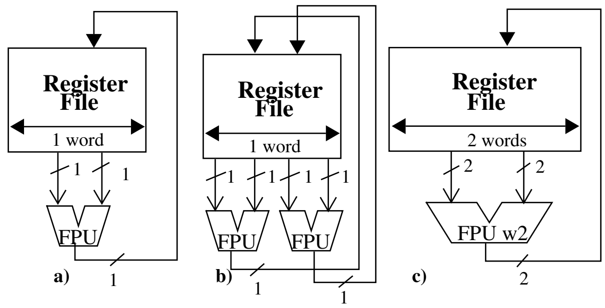


Figure 2 : Several RF and FPU configurations: a) 1-wide register file with 1 FPU, b) 1-wide register file with 2 FPUs and c) 2-wide register file with one 2-wide FPU.

the register file, the increase is linear instead of quadratic with the degree of widening. In addition the area increase of the register file in the case of widening is due to an increase of the storage capacity since the registers are also widened. In the case of replication the area grows faster while the storage capacity remains constant.

Next, we show how both techniques need some compiler support, and show the factors that limit the performance for both techniques (which is evaluated in Section 3). The evaluation is based on an estimation of the initiation interval for the software pipelined loops; it does not consider the possible impact of resource replication in cycle time. Section 5 evaluates the implementation costs in terms of area for both techniques; the evaluation considers the cost of the register file and the functional units, and does not consider the costs of other elements like memory interconnects, TLB, etc. which may have an important contribution in replication.

2.2 Limits on performance: an example

In this section we illustrate the factors that limit the exploitation of parallelism when replication and/or widening are used. Figure 3.a shows the data flow graph for the working example; nodes correspond to operations and edges represent data flow between operations. The dotted edges represent the stride between memory operations (e.g. load L0 has a stride of 1 with itself). All dependences are loop independent (i.e. have distance 0) except for the cross-iteration dependence (*5, *5) in which the result of *5 is used by the same operation *5 one iteration later (recurrence with distance one).

Figure 3.b shows the same loop unrolled, so that two consecutive iterations of the original loop are considered for scheduling.

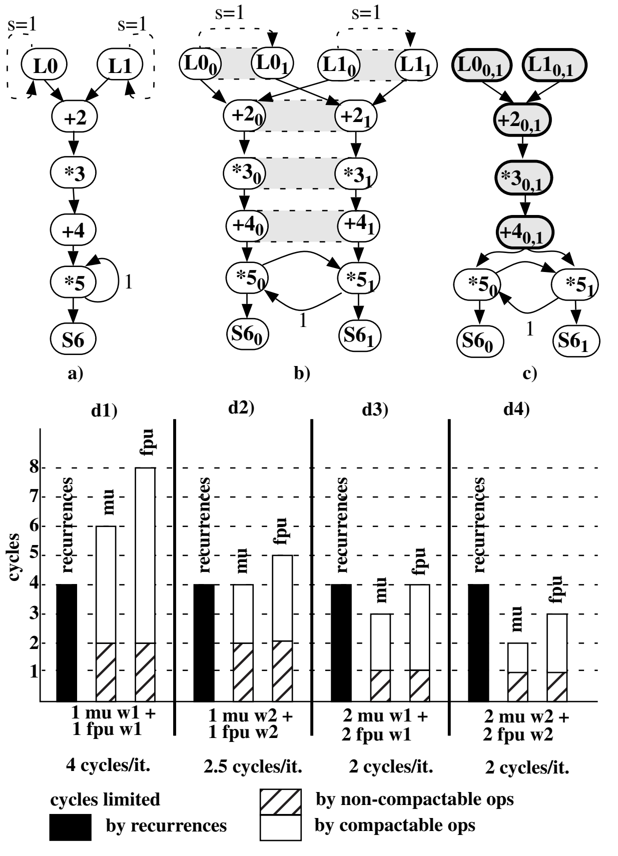


Figure 3 : a) Dep. graph for the original loop. b) Graph after unrolling (grey shadow indicates compactable ops) c) Graph with compacted ops. d) Cycles to execute two iterations for different architecture configurations.

The subscripts indicate the iteration to which the operations belong. Assume that loop 3.b is scheduled for a machine with 1 floating point unit (FPU) and 1 memory unit (MU). Assume also that all operations are fully pipelined with a latency of 2 cycles.

Software pipelining can schedule operations across the back edge of the loop, so that the only limits are the recurrences and the resources available. For any pair loop/architecture it is possible to compute the bounds of the execution time of an iteration. In our example we consider 3 separate bounds (see Figure 3.d.1):

- Recurrences: In loop 3.b there is one recurrence ($*5_0$, $*5_1$) that limits the execution time of the loop to 4 cycles per iteration, since the summation of latencies is 4.
- Memory bandwidth (MU): In loop 3.b there are 6 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 6 cycles are required to execute them.
- Execution bandwidth (FPU): In loop 3.b there are 8 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 pipelined and we only have one FPU, 8 cycles are required at least to execute them.

In conclusion, the most limiting factor of our example is the execution bandwidth that limits the performance to 8 cycles per iteration of loop 3.b (i.e. 4 cycles per iteration of the original loop).

In Figure 3.b we have joined with a gray shadow the pairs of operations that can be compacted into a single wide operation. In Figure 3.d.1 we identify, for the different resources, the cycles contributed by compactable operations (white) and the cycles contributed by non-compactable 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. Notice that the pair ($S6_0$, $S6_1$) can not be stored with a wide bus because they have a stride different than one.

Figure 3.c shows the loop after operations are compacted. This loop has 5 wide operations and 4 single operations. Figure 3.d.2 shows the execution bound when loop 3.c is scheduled for a machine with one MU and one FPU both with width two. Notice that the cycles due to the recurrence are still 4. The cycles due to resources have been reduced to 4 cycles for the memory operations and 5 cycles for the FP operations, summing up 5 cycles per iteration. Notice, however that the cycles due to non-compactable operations are not reduced by widening the functional units.

Instead of widening functional units, more parallelism could be exploited by replicating them. Figure 3.d.3 shows the execution bounds when loop 3.b is scheduled for a machine with two MUs and two FPUs. Notice that in this case we can reduce both, cycles due to compactable operations and cycles due to non-compactable operations. In this case we require 4 cycles to execute the 8 floating point operations. Another observation is that in this case we have reached the limit imposed by the recurrence and no more parallelism can be exploited.

Figure 3.d.4 shows the execution bounds when both techniques (replication and widening) are applied. Now we have a machine with two wide MUs and two wide FPUs, therefore we require 2 cycles to execute the 4 memory operations and 3 cycles (2.5 to be more precise) to execute 5 floating point operations. If this loop would not had recurrences, the loop 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 (neither by adding functional units nor by widening them).

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

- Loops with recurrences are limited by the latency of the operations involved in the recurrence even with an unbounded number of resources.

12th International Conference on Supercomputing (ICS-12), pp 441-448, Melbourne, Australia. July 13-17, 1998.

- 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 to the cycles contributed by compactable operations. Therefore the non-compactable operations limit the parallelism even with an unbounded widening degree. For an operation to be compactable it is required that it does not belong to a recurrence. In addition, for memory operations, it is also required that this operation has an access pattern with stride 1.

Figure 4.a shows a classification of loops according to the different limiting factors:

- RecB: *Recurrence bound* loops whose performance is limited by recurrences (black bar).
- Rec+OW: *Only widening with recurrences* are loops whose most limiting factor are the resource constraints; they have also recurrences whose limit can be reached only by widening the functional units. However, the same level of performance can be attained by applying a smaller degree of replication. The requirement is that the portion of the resource limit that projects over the recurrence limit corresponds to compactable operations (white part). However, the less non-compactable operations the sooner we will reach the recurrence limit by only replication.
- Rec+OR: *Only replicating with recurrences* loops are resource constrained loops with recurrences in their bodies where all the operations that require the most limiting resource are non-compactable (stripped part). The only way of increasing performance (until the recurrence limit is reached) is by resource replication.
- Rec+WbR: *Widening but replicating with recurrences* loops are resource constrained loops with recurrences where there are some compactable operations that require the most limiting resource. However, the non-compactable part (stripped part) projects over the recurrence limit, and therefore some performance benefit can be obtained by widening; however replication is also needed to achieve the maximum performance (i.e. lower the execution time until the recurrence limit is attained).
- OW: *Only widening* are loops without recurrences (i.e. vectorizable) whose operations are all compactable. Therefore the same performance can be attained either by widening or by replication. Furthermore, since there are no recurrences, the maximum performance can be attained (provided that the loop trip count is sufficiently large).
- OR: *Only replicating* loops do not include recurrences and their operations are nor compactable (at least the operations that require the most used resource). Therefore the only way to improve performance is by resource replication.
- WR: *Widening but replicating* loops are loops without recurrences that have some compactable operations and some non-compactable. Therefore some performance improvement can be attained by widening the functional units. However, replicating provides more performance benefits.

Figure 4.b shows all possible transitions between the different types of loops that produce some performance benefit. For instance a Rec+WbR loop can become a Rec+OR loop after widening the functional units. Then it can remain Rec+OR after replicating the functional units (but its performance will be closer to the recurrence limit). And, finally, it can be converted to a RecB loop after applying replication again, achieving the maximum possible performance for this loop. Once the loop is RecB, its performance will remain constant despite of the number and width of additional functional units.

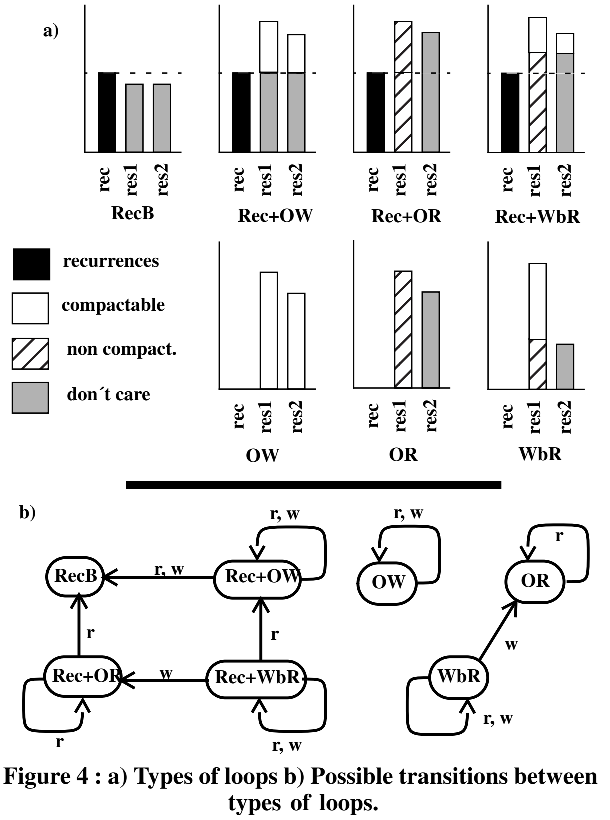


Figure 4 : a) Types of loops b) Possible transitions between types of loops.

3. Performance evaluation

This section analyzes the impact on performance of replication and widening applied to memory access ports and functional units. First of all, the experimental framework and processor configurations are described. Then the configurations are evaluated without taking into account implementation costs.

3.1 Experimental framework

The experimental evaluation in this paper is based on the following components: the dependence graphs for a collection of loops taken from the Perfect Club benchmarks [2], a set of processor configurations with varying memory bandwidth and computing capabilities, and a performance estimation module for software pipelined loops:

- A total of 1180 innermost loops from the Perfect Club benchmarks, all suitable for applying software pipelining and representing a high percentage (around 78%) of the total execution time. They have been generated by the ICTINEO [1] compiler and are annotated with some static and dynamic information (such as iteration trip, execution time, strides in memory accesses, ...)
- A set of VLIW processor configurations with varying number and width of general purpose functional units and bidirectional memory access ports. In most configurations, it is assumed that the number of functional units doubles the number of memory ports. In general, configuration xwy has x memory ports capable of accessing y-wide words, and $2x$ functional units able to compute with y-wide words. The values for x and y used along the paper are 1, 2, 4, 8, and 16. For instance, configuration *4*w*2* includes 4 memory ports and 8 functional units, all of them able to operate with operands of width equal to 2 words.
- The performance estimation module applies the appropriate degree of unrolling to each dependence graph and a compaction algorithm that groups nodes of the unrolled dependence graph in order to detect wide operations (both memory accesses with stride 1 and arithmetic operations). The module estimates the execution time of a loop on a specific machine configuration as

the product of its minimum initiation interval[1] *MII=max(RecMII, ResMII)* and the iteration trip (number of iterations) of the loop.

Table 1 characterizes each program in the benchmark showing a breakdown of the loops according to the previous classification.

Program	% exec. time in loops with recurrences					% exec. time in loops without rec.			
		RecB	R+OW	R+OR	R+WbR		OW	OR	WbR
ADM	50	20	6	22	2	50	40	0	10
QCD	56	0	1	0	55	44	43	1	0
MDG	39	1	37	0	1	61	20	0	41
TRACK	44	1	8	1	34	56	55	0	1
BDNA	99	4	0	16	79	1	1	0	0
OCEAN	74	2	6	3	63	26	8	1	17
DYFESM	28	19	1	1	7	72	70	1	1
MG3D	79	1	0	78	0	21	18	1	2
ARC2D	5	1	3	1	0	95	67	5	23
FLO52	3	0	3	0	0	97	86	1	10
TRFD	5	0	1	1	3	95	1	0	94
SPEC77	69	2	2	49	16	31	6	7	18
ALL	66	1	4	52	9	34	23	1	10

Table 1: Execution time spent in the different types of loops.

3.2 Performance and limits

In this section we analyze the impact on performance of replication and widening applied to memory access ports and functional units. The conclusions drawn from these experiments are independent of implementation related aspects and only analyze the impact on performance of these techniques. In the next two sections we take into account technology trends and restrictions in order to relate performance gains and implementation costs.

Figure 5 summarizes the impact on performance of replication and widening applied to the two components, memory and functional units. The horizontal axis in this figure represents the total memory bandwidth (b=$x * y$) for a machine configuration xwy. :

The lower plot represents the application of bus widening. All configurations have a single memory port (*1*w*b*). Notice that this plot saturates very fast towards a maximum speed-up a little bit higher than three. This is due to the fact that only a reduced portion of operations (memory accesses or arithmetic) benefit from high degrees of compaction (stride 1 required). Last row in Table 1 supports this statement: 27% of the loops (columns R+OW and OW) directly benefit from widening. The rest of them are limited (or can become limited) by recurrences or need replication.

The upper plot represents the application of bus replication alone. All configurations have width equal to 1 (*b*w*1*). Notice that these configurations perform better than the previous ones because they are less restrictive (although more expensive); a large number of loops in the benchmarks evaluated benefit from increasing the number of buses and functional units. Again, the last row in Table 1 supports this statement: 72% of the loops (columns R+OR, R+WbR, OR and WbR) benefit from replication.

Intermediate points between the two previous plots reflect the impact of different degrees of replication and widening. Notice that for a specific memory bandwidth, applying small degrees of widening results in a small reduction of performance. For instance

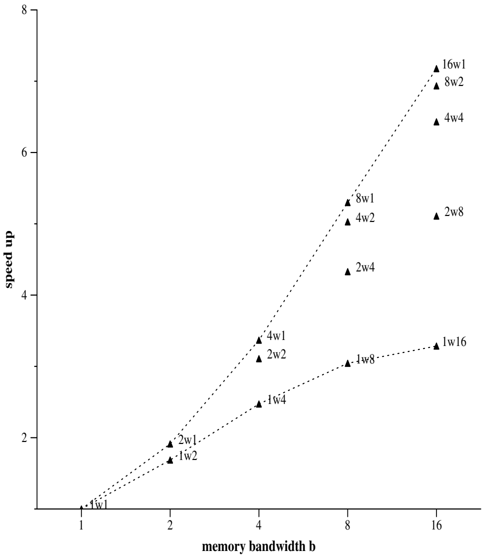


Figure 5 : Speed-up for different configurations xwy.

configurations *2*w*2* and *4*w*2* attain an slowdown of 0.92 and 0.95, respectively, when compared with their counterparts *4*w*1* and *8*w*1*, respectively. However, applying higher degrees of widening degrades performance very rapidly.

In general, diminishing returns are obtained when bandwidth *b* is increased, either with replication (upper plot) or widening (lower plot). The diminishing returns of replication are due to the fact that increasing bandwidth (both for functional units and memory accesses) converts resource bounded loops into recurrence bounded loops; once a high percentage of loops is limited by recurrences, increasing bandwidth only affects to fully vectorizable loops. The diminishing returns of widening are due to a lack of compactable operations, specially memory accesses that require stride one).

In order to identify the individual contribution of the techniques applied to memory access ports or functional units, we have conducted two additional experiments, whose results are shown in Figure 6. Figure 6.a shows the impact on performance on all the loops evaluated when varying the number of buses and their width (configurations xwy), but assuming *2x* functional units of width 1. This figure is quite similar to the general one, so the same conclusions above can be drawn. Figure 6.b shows the impact on performance on all the loops when varying the number and width of the fp units, but assuming that all memory ports have a width equal to 1. Notice that now, the performance of all configurations with the same number of issued operations per cycle is quite similar. In other words, applying widening results in almost the same performance than applying replication. This is due to the high degree of compaction that can be applied to floating point operations in numerical programs. The main difference when compacting memory accesses is that they require stride 1 accesses to memory; this constrain is not needed when compacting computations.

In the next sections we analyze the impact on performance of both techniques taking into account implementation costs. We conclude that applying small degrees of widening clearly return similar performance gains than replication alone, but with a considerable smaller cost.

4. Technology projection

In this section we estimate the area cost of the different configurations evaluated. For this purpose, we take into account the SIA *Semiconductor Industry Association* predictions [16] for the technology size (λ) and the chip size in order to compute the number of

1.Current modulo scheduling proposals achieve near optimal schedules in terms of II (for instance HRMS [10] obtains the *MII* in 97% of the loops that compose our benchmarks platform). Therefore, the *MII* can be considered a very good metric for estimating the *II* after scheduling.

λ^2 per chip for the next five generations (Table 2).

	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

Table 2: Semiconductor Industry Association (SIA) predictions in 1994

We have estimated the area of a general purpose floating point unit (FPU) using the one included in the MIPS R10000 processor as a reference. The R10000 processor FPU can execute two floating-point operations simultaneously and 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 μm technology, the R10000 FPU requires 12 mm^2 of area [13]. So we assume the area of a FPU as 12 mm^2 x 16x10^6 λ^2/mm^2 = 192x10^6 λ^2.

Besides, the overall size of a register file is determined mainly by the size of a register cell, the most replicated part of the RF. Other components that are needed to access the register file, such as decoders and read/write drivers for the data lines, typically represent less than 5% of the area required by the register cells [9].

Figure 7 shows a register cell with one read and one write port. This cell has been implemented using scalable CMOS [9]. To access the register cell, 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 grow both the height and the width. The memory portion of the register cell is a pair of cross-coupled inverters consisting of four transistors that force a minimum height of 41λ. The memory portion of the cell that can accommodate 3

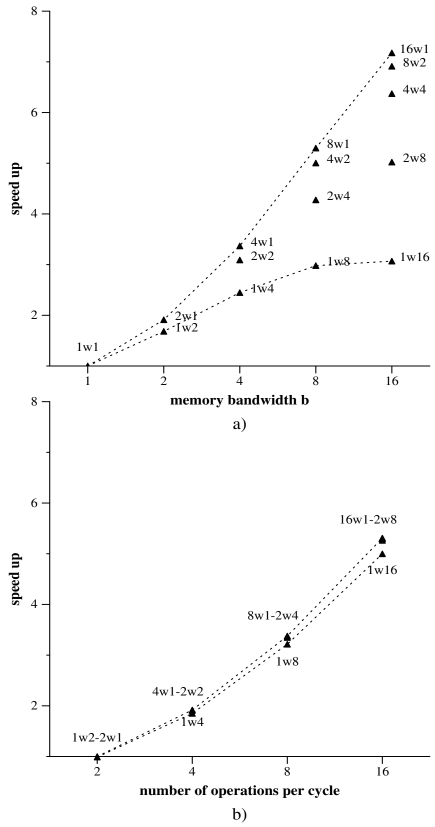


Figure 6 : Speed-up for different configurations xwy: a) with constant width for computation operations (one), and b) with constant width for memory accesses (also one).

select lines running width-wise across the cell. Therefore, the height of the cell does not grow until more than 3 ports are implemented. After that, each pots adds 8λ to the height.

The width of the dual ported cell above 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. Table 3 shows the dimensions of several multiported register cells.

Ports	1R, 1W	2R, 1W	4R, 2W	5R, 3W	8R, 5W	10R, 6W
W x H	50x41	64x41	120x65	162x81	260x121	316x145
Area (λ^2)	2050	2624	7800	13122	31460	45820
Relative ar.	1	1.28	3.80	6.4	15.35	22.35

Table 3: Dimensions of several multiported register cells.

In order to better understand how those factors influence the total area cost, let´s consider our baseline processor *(1w1)*. The memory bus requires one read and one write port to the register file, and each functional unit requires 2 read ports and one write port. Therefore, the *1w1* configuration has a register file (RF) with 5 read ports and 3 write ports. From table 3, such configuration would require 13122 λ^2 per bit. Assuming a 64-bit 64-registers RF, *1w1* requires 53.75x10^6 λ^2 for the RF and 384x10^6 λ^2 for the functional units.

If we double the number of functional units, i.e. a configuration with 2 memory access and 4 functional units (either *2w1* or *1w2*) the area required for the functional units is twice the area required by *1w1* (i.e. 768x10^6 λ^2) independently of the way of doubling the number of functional units (by replicating them or by widening them). However, if we replicate the functional units (configuration *2w1*) we have a 64-bit 64-registers RF with 10 read ports and 6 write ports. From Table 3, this register file requires 45820 λ^2 per bit, with a total RF area of 187.7x10^6 λ^2 (3.49 times more area for doubling the number of ports). On the other hand, if we widen the functional units (configuration *1w2*) we have a 128-bit (two 64 bit words per register) 64-registers RF with only 5 read ports and 3 write ports. This register file has the same area per bit and twice the number of bits than the *1w1* configuration, resulting in a total register file area of 107.5x10^6 λ^2 (2 times more area for doubling the wide of the ports). Therefore the *1w2* configuration has the same number of functional units as the *2w1* configuration but requiring only 57.3% of the register file area with the additional benefit that its register file can store twice the number of words.

In the previous example we considered moderated configurations and therefore the dominant part of the area were the functional units. Considering both, functional units and register file, the *1w1* configuration requires approximately 474x10^6 λ^2, *2w1* requires 956x10^6 λ^2, and *1w2* requires 875x10^6 λ^2 (91.5% of the total area required by *1w2*). However, when more aggressive configurations are considered, the dominant part becomes the register file since its area grows (for a large number of ports) quadratically

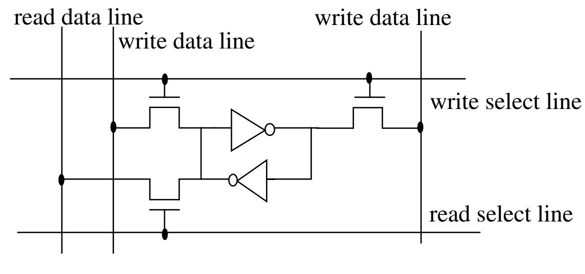
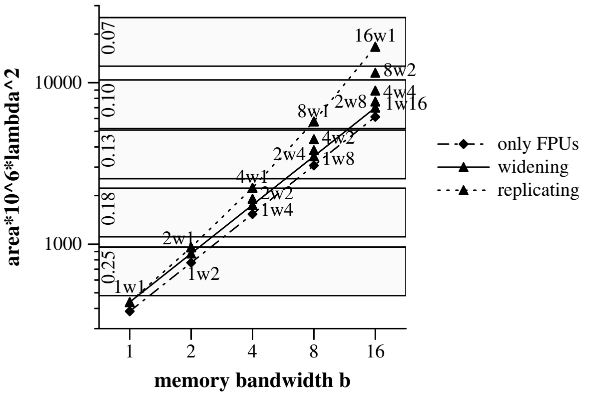
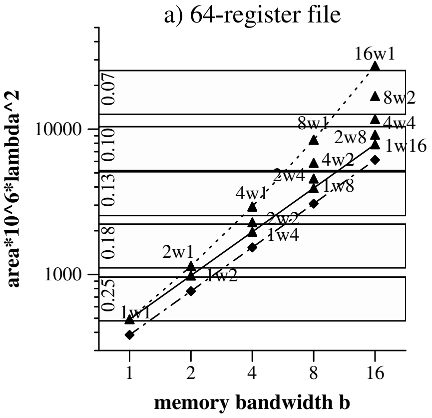


Figure 7 : Schematic diagram of a 1 read, 1 write registre cell using scalable CMOS.



a) 64-register file



b) 128-register file

Figure 8 : Size of the reg. file and floating point units for different configurations xwy a) 64 and b) 128 registers RF.

with the number of ports. For instance, configuration *16w1* with a 64-bit 256-registers RF requires 48306x10^6 λ^2 while configuration *1w16* with 1024-bit (sixteen 64 bit words per register) 256-registers RF requires only 9584x10^6 λ^2. In other words, configuration *1w16* has the same number of functional units as *16w1*, sixteen times more storage capacity in the register file since each register is 16 words wide and requires only 19.8% the area of *16w1*.

Figure 8 summarizes the total area cost of replication and widening applied to the two components, memory and functional units, for configurations 64 and 128 registers RF. The horizontal axis in this figures represents the total memory bandwidth (b equals *x* times *y*) for a machine configuration *x*w*y*. The width of the registers varies with the width of the functional units, therefore it must be noticed that wide configurations have more storage capacity.

The lower plot (diamonds) represents the area due to the functional units. The solid plot represents the application of widening. All configurations have a single memory port (*1*w*b*). Notice that the area required grows linearly with the degree of widening. The upper plot (dotted line with triangles) represents the application of replication alone. All configurations have width equal to 1 (*b*w*1*). Intermediate points between the two previous plots reflect the impact of different degrees of replication and widening. Notice that the more aggressive the configurations (more functional units and more registers) the bigger the impact of the register file in the total area when replicating is applied (observe that the vertical axis has logarithmic scale).

We assume that it is reasonable to use between 10% to 20% of the chip area to the functional units and the register file. The horizontal bands in Figure 8 represent the chip area between 10% (lower side) and 20% (upper side) of the total chip area for the five SIA predictions in Table 2. These bands represent what would be the most expansive configurations that could be implemented with a given technology. For instance in the year 2010 it will be possible to implement a *2w8* or a *8w1* configuration with a 128-registers RF devoting at most 20% of area to implement the register file and the functional units. If only 10% of the chip is devoted to these two components, then it will be possible to implement configurations like *4w1* or *2w4*, but not *2w8* and *8w1*.

5. Performance/cost trade-off

In the previous sections the performance and cost of the widening and replicating techniques have been studied. Before trading-off both aspects, we draw some preliminary conclusions:

- Doubling the number of operations that can be done per cycle in a given configuration using replication always obtains better performance than using widening.
- For both techniques, every time the number of operations that can be done per cycle increases, the relative increment of performance between the old configuration and the new configuration is smaller. This is because of in a more aggressive configuration, the number of loops that achieve the maximum parallelism is bigger than in a less aggressive configuration.
- Doubling a configuration by using widening doubles the cost, while doubling a configuration by using replication has an increment of cost that grows quadratically respect to the number of resources.

Figure 9 shows the performance and cost of the different configurations tested, for register files of 32, 64, 128, and 256 registers. Looking at the results, one can notice that using widening increases the performance more than the cost. For example, an architecture with 8 memory accesses and 16 fp operations per cycle can be implemented with configurations *8w1* and *4w2*. The *4w2* configuration achieves 94.9% of performance of the *8w1* configuration at a fraction of its cost (62.6% in a 256-registers RF).

Another important point is that with only replication, there is a enormous cost gap between one configuration and the following one. For example, using a 128-registers RF and assuming that the area of the RF plus the area of the FPUs must be between 10% and 20% of the total chip area, configuration *1w1* can be implemented with a 0.25μ technology. The next configuration *(2w1)* can not be implemented with a 0.25μ technology, but it can with a 0.18μ. The other configurations (*4w1*, *8w1* and *16w1*) require 0.13μ, 0.1μ and 0.07μ technologies to be implemented, respectively.

Some configurations that use replication and widening have a cost between two configurations that only use replication. If we have a technology of 0.13μ and need a 128-registers RF, configuration *4w1* can be implemented; however, configuration *8w1* cannot be implemented. But we can implement *2w4* which has a relative increment of performance of 1.28 with respect to *4w1*. We conclude that having an area limitation, mixing replicating and widening can provide the configuration that achieves maximum speed-up with a cost between the given limits.

Finally, we want to remark that the performance results shown in this study have been estimated assuming an infinite number of registers. When the number of registers is limited, some values must be temporarily stored in memory and accessed with spill code. In this case, we can take advantage of widening. For instance, configuration *8w1* with a 128-registers RF and configuration *4w2* with a 256-registers RF can be both implemented with a 0.10μ technology. In terms of performance, *4w2* achieves 94.9% of the performance of *8w1*. However, for a similar cost, *8w1* has only 128 single registers (64 bits), while *4w2* has 256 wide registers (2 64 bit words per register) with 4 times the storage capacity. If spill code is required, it is expected (as some preliminary results show) that *4w2* would achieve better performance than *8w1*. The evaluation of this aspect is part of our current work.

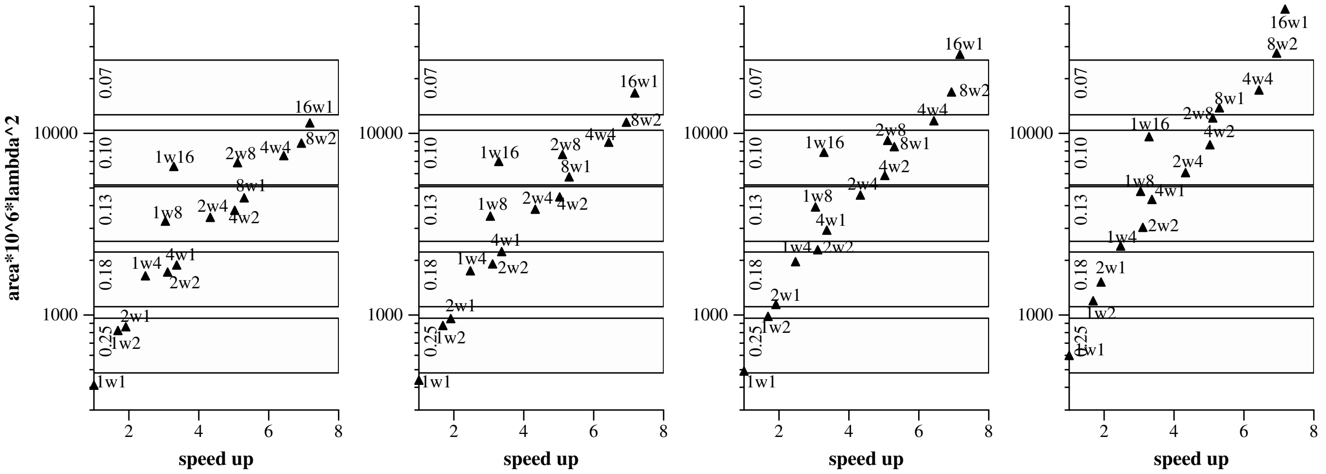


Figure 1 : Performance/cost trade-off for different configurations xwy, assuming a) 32 b) 64 c) 128 and d) 256 registers RF

6. Conclusions

In this paper we have focused on trading-off performance and implementation costs for different design alternatives oriented towards having high memory bandwidth and computational capabilities in near future high-performance architectures.

Resource replication and widening are the two techniques used to accomplish this objective. Widening resources (either memory access ports or functional units) should provide the same peak bandwidth than replicating resources, but at a fraction of the cost (area and access time). However, an efficient exploitation of widening requires certain characteristics in the programs and compiler technology. For instance, well-known vectorizing techniques could be used to detect wide memory accesses (a wide access can be seen as a stride-one vector memory access with a short vector length) and wide computations; however, it is necessary to have additional compilation techniques that compact independent memory accesses and operations in addition to vectorizable code.

We have analyzed the impact on performance of both techniques taking into account implementation costs. We have concluded that widening alone is not sufficient to attain the memory bandwidth and computing capabilities required by numerical applications. However, applying small degrees of widening to a configuration which uses replication clearly return similar performance gains than replication alone, but with a considerable smaller cost. For instance, configuration *4w2* achieves 94.9% of performance of *8w1* with a 62.6% of cost (with a 256-registers RF)

Resource widening imposes more constraints to the software pipelining scheduler because compacted operations have to be scheduled at the same cycle. This may slightly increase the loops register requirements. However, widening also implies an increase in the total capacity of the register file; this additional capacity will counteract amply the slight increase in the register pressure,thus resulting in lower register pressure. Preliminary results show that when performance degradation of spill code is considered, widening produces performance results closer to replication.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIC 429/95.

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. RR UPC-DAC-1996-61. Dec 1996.

[2] M. Berry, D. Chen, P. Koss and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. TR 827, CSRD, University of Illinois at Urbana-Champaign, Nov. 1988.

[3] D.C. Chang, D. Lyon, C. Chen, L. Peng, M. Massoumi, M. Hakimi, S. Iyengar, E. Li and R. Remedios. Microarchitecture of HAL's memory management unit. *CompCon95,* pp 272-279, 1995.

[4] J.C. Dehnert and R.A. Towle. Compiling for Cydra 5. *Journal of Supercomputing,* 7(1/2):181-227, 1993.

[5] J.H. Edmonson, P. Rubinfeld, R. Preston and V. Rajagopalan. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, 15(2):33-43, April 1995.

[6] R.B. Jones and V.H. Allan. Software pipelining: A comparison and improvement. In *Proc. of MICRO-23*: 46-56, 1990.

[7] R. Jolly. A 9-ns 1.4 gigabyte 17-ported CMOS register file. *IEEE J. of Solid-State Circuits*, 25 :1407-1412, Oct. 1991.

[8] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc.* of *PLDI* pp 318-328, 1988.

[9] C. G. Lee. *Code Optimizers and Register Organizations for Vector Architectures*. Ph.D. Thesis U.C. Berkeley, May 1992.

[10] J. Llosa, M. Valero, E. Ayguadé and A. González. Hypernode reduction modulo scheduling. In *Proc. of the Micro-28* pp 350-360 , 1995

[11] 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.

[12] D. López, M. Valero, J. Llosa and E. Ayguadé. Increasing performance with multiply-add units and wide buses. Research Report UPC-DAC-1997-80. 1997.

[13] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson and K. Chang. The case for a single-chip multiprocessor. In *Proc. of the ASPLOS-VII*, pp 2-11, 1996

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

[15] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the Micro-27*:63-74, Nov. 1994.

[16] The National Technology Roadmap for Semiconductors, Semiconductor Industry Assoc., San Jose, Calif. 1994

[17] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proc. of the ASPLOS-IV* :53-62, 1991.

[18] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Publishing, 1988.