Increasing Memory Bandwidth with Wide Buses: Compiler, Hardware and Performance Trade-offs

David López, Mateo Valero, Josep Llosa and Eduard Ayguadé
{david | mateo | josepll | 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
Memory latency and lack of bandwidth are the main barriers to achieve high performance from current and future processors, specially in numeric applications. New organizations of the memory subsystem as well as hardware and software mechanisms to effectively exploit them are required. The paper presents a new compilation technique to pack several load/stores (that access consecutive memory locations) into a single wide load/store, so that the number of wide load/stores is maximized. It also evaluates the performance trade-offs of wide buses and the additional register pressure, showing that it is minimal and has negligible effects. Finally, the paper proposes a hardware mechanism to detect and group memory accesses into wide accesses at run time, so that binary compatibility is preserved. The evaluations are performed using 1180 loops that represent about 78% of the execution time of the Perfect Club. The results reveal that using wide buses is a cost-effective solution to improve the bandwidth between the processor and the first-level cache.

1. INTRODUCTION
Microprocessor performance is increasing at higher rates than the performance of the memory subsystem. High-performance microprocessors make use of pipelining and parallelism in order to have a shorter cycle time and maximize the work done per cycle (i.e. multiple instruction execution). To effectively exploit instruction level parallelism (ILP), superscalar processors rely on architectural techniques (dynamic instruction execution), compilation techniques (aggressive scheduling techniques such as software pipelining) or a combination of both. These techniques increase the efficiency of the processor core, requiring low latency and high bandwidth memory subsystems. 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 has large memory bandwidth demands particularly in numeric applications. To meet the high bandwidth requirements, some current microprocessors have been built with two memory buses [WhDh94, Hsu94, ERPR95, Hun95]. Doubling the number of buses has several costs and drawbacks at different levels of the memory hierarchy:

- Processor core: It requires doubling the number of load/store units and additional register-file ports [GAB+88, LMM92], that increase the access time [WeEs88]. Furthermore there is a noticeable area penalty since, in CMOS, the area of a multi-ported device is proportional to the square of the number of ports [Jol91].
- On-chip cache: A dual-ported cache is required to allow two memory accesses per cycle. This cache can be implemented by duplicating data cache and maintaining identical copies of data in each cache [ERPR95] that doubles the die area of the primary data cache, or by splitting the data cache in two (or more) independent banks [SoFr91]. However, bank conflicts can reduce the effective bandwidth.
- Translation Lookaside Buffer: Multiporting the TLB can increase cycle time as well as require some extra die area. Some processor designs implement multi-level TLBs [CLC+95]. However, a single port TLB will provide the same, or even better, hit rate requiring less area.
- Off-chip connection: Performing two off-chip memory accesses per cycle, requires two buses with address, control and data information. Having two off-chip buses, increases the number of pins required (complicating and making expensive the package of the chip) and increases the complexity of the off-chip memory system (similar problems to on-chip caches).

An approach to have high bandwidth with reduced cost is to increase the width of the memory buses and move several (consecutive) words per memory access. In [WOR96], several hardware techniques for exploiting the bandwidth of a single wide port are proposed. However, these techniques, are only applicable to out-of-order superscalar processors. Moreover, the techniques still require the generation of several addresses per cycle, additional buffering to hold pending loads/stores and additional hardware to bind incoming data to pending memory accesses.

An alternative to exploit wide buses is to have, at the instruction set level, explicit wide load/stores as in the POWER2 microprocessors [WhDh94]. This alternative has the advantage that, to perform two loads/stores per cycle, only one issue slot, one address generator, one address translation per cycle, and one wide (i.e. one address, one control and two data buses) memory port are required. Furthermore, it can be used with any kind of processor (dynamic and static superscalar processors, and even VLIW). Nevertheless, this technique has also some drawbacks. Wide loads/stores must be compacted statically by the compiler, losing some potential performance gains in programs with irregular access patterns. They also require some new instructions in the processor, which break backward compatibility with previous implementations of the architecture.

This paper presents a compilation technique, based on a dependence graph extended with stride information, to pack two (or more) single loads/stores into one wide load/store. The paper evaluates the performance benefit of a wide bus (Figure 1c) versus a single bus (Figure 1a) and the performance degradation in front of two buses (Figure 1b). For the evaluations, we use 1180 loops that account for 78% of the execution time of the Perfect Club [BCKK88]. Even though having wide loads/stores can be applied

11th International Conference on Supercomputing (ICS-11), pp 12-19, Vienna, Austria. July 7-11, 1997.

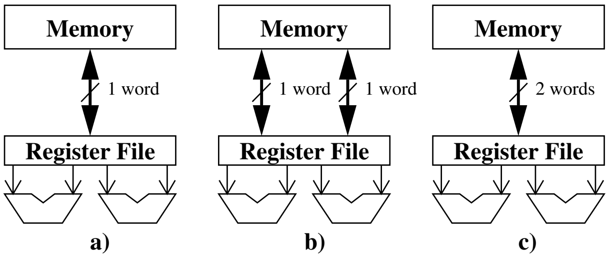

Figure 1: Bus configurations studied: a) one single bus, b) two single buses, c) one wide bus.

to both dynamically and statically scheduled machines, the paper is targeted to statically scheduled machines. We use an aggressive scheduling technique to obtain the maximum performance and stress the memory system to maximum. The scheduling technique used is Hypernode Reduction Modulo Scheduling *HRMS* [LVAG95], a software pipelining technique that tries to produce throughput optimal schedules with reduced register requirements.

Wide buses require the detection, at compilation time, of *compactable* load/stores (i.e. independent memory accesses to consecutive memory locations). Most of these accesses come from sequential accesses to arrays so unrolling may be necessary in order to pack instances of the same load/store from different iterations into the same wide access. We evaluate the performance lost of a wide bus in front of two single buses. In addition, some current processors have already been implemented with two memory buses, however to maintain the exploitable ILP of future designs, more memory bandwidth will be required. Since going to four memory buses will be difficult, we evaluate the improvement offered by configurations with two wide buses.

A drawback of wide loads/stores is that two independent load/store operations that have been compacted together are forced to be scheduled together; this may enlarge the lifetime of values that came from or go to memory, thus increasing the register requirements. However, we show that the increase in the register requirements is small. Besides, the increment in register requirements is, in general, produced in small loops, which require few registers, and therefore will not require the addition of spill code.

Backward compatibility can be of high interest in future implementations of an existing processor family. For instance, code compiled for a POWER2 can not be executed in a RS-6000 [IBM94]. This paper proposes a hardware technique that fuses consecutive loads/stores before issuing them, so that 2 compactable loads/stores progress through the pipeline as a single wide load/store. This technique still requires compiler support to obtain good performance levels, but the code is fully compatible in any direction (forward and backward). We show that with this option we can achieve 97% (in the worst case) of the performance of a machine with wide load/store instructions without losing code compatibility.

The outline of the paper is as follows. Section 2 presents an overview related concepts. In section 3, an example is used to illustrate advantages and drawbacks of the use of wide buses. Section 4 describes how to compact memory accesses at compile time. Section 5 shows the performance of different bus configurations. In section 6, the problem is focused on superscalar processors and a mechanism is presented to use wide buses without lost of binary compatibility. Section 7 states our conclusions.

2. OVERVIEW OF RELATED CONCEPTS

2.1 Software Pipelining and Modulo Scheduling.

Software pipelining is a compilation technique to extract ILP from innermost loops. Modulo Scheduling [RaGl81] is a family of software pipelining techniques. Since modulo scheduling has high register requirements [MAD92,LVAL94], some scheduling algorithms have been proposed to generate near optimal throughput schedules with low register requirements, as Huff's Slack Scheduling [Huf93] and HRMS [LVAG95].

In a software pipelined loop the schedule for an iteration is divided into stages so that the execution of consecutive iterations which are in distinct stages is overlapped. The number of stages in one iteration is termed **stage count** (SC). The number of cycles between the initiation of successive iterations (i.e. the number of cycles per stage) in a software pipelined schedule is termed the **Initiation Interval** (*II*) [RaGl81].

The Initiation Interval *II* between two successive iterations is bounded either by recurrences in the graph (*RecMII*) or by resource constrains of the architecture (*ResMII*). This lower bound on the *II* is termed the **Minimum Initiation Interval** (*MII*). The reader is referred to [DeTo93, Rau94] for an extensive discussion of how to calculate *ResMII* and *RecMII*.

The execution record of the loop can be divided in three phases: a ramp up phase that fills the software pipeline (termed the prologue), an steady phase where the software pipeline achieves maximum overlap of iterations (where a piece of code, termed the kernel, is iterated), and a ramp down phase that drains the software pipeline (termed the epilogue).

2.2 Loop unrolling.

Unrolling replicates the body of a loop with step *s* a number of times (called the **unrolling factor** *u*), and then iterates by step *u times s* instead of the original step. Unrolling allows a better usage of resources because several iterations can be scheduled together increasing the number of instructions available to the scheduler. It also reduces the loop overhead caused by branching and index update. A study of the benefits of unrolling on several architectures can be found elsewhere [DoHi79]. In our case we achieve efficient iteration overlapping through Software Pipelining. Anyway 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 fractional *MII* [JoAl90]. Also, in [LaHw95] there were proposed a set of software pipelining based optimizations that benefit from unrolling. In this paper, unrolling allows us to compact (if possible) memory accesses in consecutive iterations of the loop.

2.3 Register Requirements

Each value in a loop is alive for a certain amount of cycles termed *lifetime*. Lifetimes in loops correspond either to *loop-invariant* variables or to *loop-variant* variables. Loop-invariant variables are repeatedly used but never modified during the loop execution. Loop invariants are produced before entering the loop and are alive during all the execution of the loop, requiring one register each one during the execution of the loop. For each loop-variant, a new value is produced in each iteration of the loop and, therefore, there is a different lifetime corresponding to each iteration. Because of the nature of software pipelining, lifetimes of values defined in an iteration can overlap with lifetimes of values defined in subsequent iterations. A lower bound of the register pressure of the loops is obtained by computing the maximum number of values that are alive at any cycle of the schedule. Lifetimes of loop variants start when the producer is issued and end when the last consumer is issued. How to allocate registers on software pipelined loops is beyond the scope of this paper; for an extensive discussion of the problem see [RLTS92]. Nevertheless, some of these techniques almost always achieve the lower bound; in the paper we use the lower bound instead of actual register requirements.

3. MOTIVATING EXAMPLE

As an example consider a processor with one adder and one multiplier for which we will consider the distinct bus configurations shown in Figure 1. Assume that both, the adder and the multiplier, are fully pipelined, with a latency of 2 cycles, and that load/store

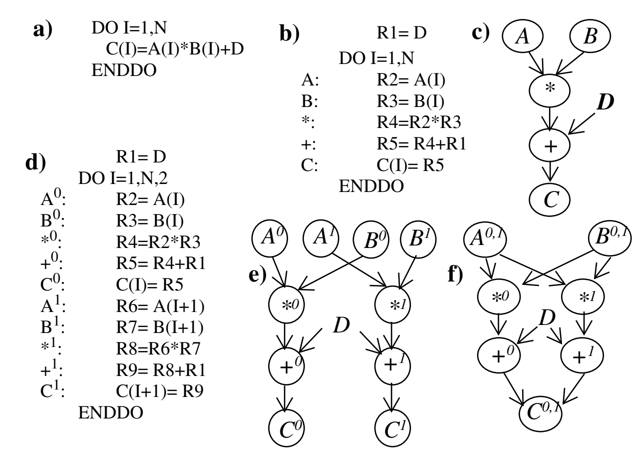

Figure 2: (a) Example loop. (b) Optimized pseudo-code representing the operations in the loop and (c) its dependence graph. (d) Optimized pseudo-code with unrolling factor=2 and its dependence graph with(f) and without(e) wide buses

operations can be served in 1 cycle.

Consider the loop shown in Figure 2a. Figure 2b shows the pseudo-code of the loop representing the operations in the loop body and Figure 2c shows the associated dependence graph. Figure 2d shows the loop unrolled with u=2 and Figure 2e the associated dependence graph. In these figures, the superscripts associated to operations indicate to which replication of the original loop they belong to (0..u-1).

Figure 3a shows a schedule for one iteration of the unrolled loop for an architecture with one single bidirectional bus (see Figure 1a). It also shows the lifetime of the variables involved. Each value is marked with the operation that produces it. Figure 3b shows the kernel code of the scheduling, where the subscripts indicate the stage of the schedule in which the operation has been scheduled. In this case, 6 cycles are needed to execute two iterations (3 cycles per iteration), since the most used resource is the load/store unit and there are 3 memory accesses per iteration. This schedule requires at least 5 registers since there are up to 5 variables overlapping at a given cycle (rightmost column in each diagram).

Figure 3c shows the schedule of the loop with 2 buses (see Figure 1b). Notice that 2 iterations can be scheduled in 3 cycles (1.5 cycles per iteration). In addition, reducing the *II* increases the register pressure. For instance, in this case 6 registers are needed.

In this example all the memory accesses have stride 1. With a single wide bus (see Figure 1c), two memory accesses that access consecutive memory locations (for instance A^0 and A^1) can be performed simultaneously. In this case, the unrolled graph (Figure 2e) must be compacted, resulting the graph shown in Figure 2f. Figure 3d shows a schedule of the compacted graph with one wide bus. Notice that the schedule achieves the same throughput than having two buses. Because two compacted accesses must be done together, the use of this technique can increase the lifetime of the variables and therefore the register requirements (in the example, 9 registers are required).

Wide buses can achieve, in the best case, the same performance as multiple buses. However, if operations with stride 1 are not available or they can not be compacted, the maximum performance can not be achieved. In addition, wide buses may have the penalty of an increase of registers requirements. In the following sections we study how to perform the compaction of memory accesses at compile time, and evaluate the performance and register requirements of wide buses.

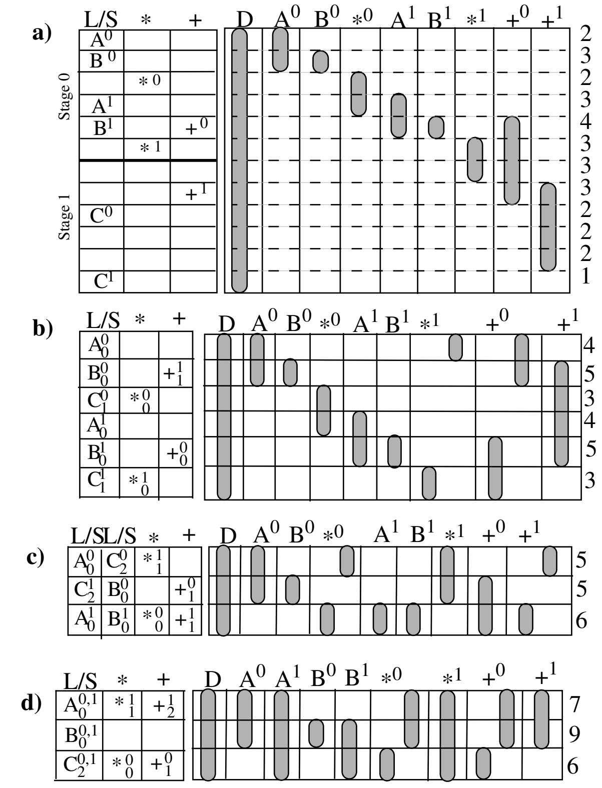

Figure 3: Scheduling and registers requirements of the unrolled loop with one single bus applying software pipelining a) of a single iteration and b) of the kernel code. c) the same loop with two buses configuration and d) its scheduling and registers requirements with a wide bus.

4. MEMORY ACCESS COMPACTION

To benefit from wide buses, memory accesses with stride 1 are required. Next we define dependence graphs that include information about strides and an algorithm to detect memory accesses with stride 1. Also the algorithm to compact these accesses is described.

4.1 Dependence Graph and Extended Dependence Graph

The dependences of an innermost loop can be represented by a Dependence Graph $G = DG(V,E,\delta)$. V is the set of vertices of the graph G, where each vertex $v \in V$ represents an operation of the loop. 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 non-negative 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.

For this study, we define the *Extended Dependence Graph* $G' = EDG(G,S,\sigma)$ that includes information about strides. G is the dependence graph previously defined. 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 of data involved in operation u and v. This difference is the stride $\sigma_{(u,v)}$ associated to the edge (u,v).

If the dependence graph has been calculated before unrolling the loop (as most of compilers do) new stride 1 edges can be deduced. Figure 4 shows an example where a new stride edge is deduced when unrolling the *EDG*. Analysing the original code

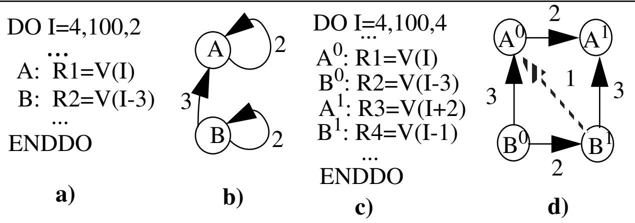

Figure 4: a) Original code and b) its associated EDG (only stride edges). c) The same code with unrolling factor u=2 and d) its associated EDG with the deduced stride edge (dashed arrow).

(Figure 4a) no stride 1 is detected, but in the unrolled code (Figure 4c) a stride 1 edge (from B^1 to A^0) is clearly defined. The main idea of the algorithm is that if one node *w* has two stride edges as input (or as output) from (to) *u* and *v* with different σ then another stride edge exists between *u* and *v*. If two incoming (outgoing) edges from (to) a node have the same σ, then the two source (target) nodes access the same memory position. However, since redundant loads/stores have been previously eliminated, this situation can not occur.

The algorithm to deduce new stride edges is as follows:

Let A, B and C be three memory operations with the same behavior (three loads or three stores to the same non-scalar variable)
case 1: /* a node with two outgoing edges to different nodes */
there is an stride edge (A,B) (i.e. from A to B) with $\sigma_{(A,B)}$=m
and another stride edge (A,C) with $\sigma_{(A,C)}$=n {
case (m>n) add new stride edge (C,B), $\sigma_{(C,B)}$=m-n if it doesn't exist.
case (m=n): not possible (the graph has been previously simplified).
case (m<n) add new stride edge (B,C), $\sigma_{(B,C)}$=n-m if it doesn't exist.
}
case 2:/* a node with two incoming edges from different nodes */
there is an stride edge (A,C) with $\sigma_{(A,C)}$=m
and another stride edge (B,C) with $\sigma_{(B,C)}$=n {
case (m>n): add new stride edge (A,B), $\sigma_{(A,B)}$=m-n if it doesn't exist.
case (m=n): not possible (the graph has been previously simplified).
case (m<n): add new stride edge (B,A), $\sigma_{(B,A)}$=n-m if it doesn't exist.
}

This is an iterative algorithm that must be applied to all nodes in *G'* that match the condition, until no new edges are added to *G'*.

4.2 Compaction Algorithm

We define **compactable accesses** as a number of **w** memory accesses that can be performed in a single memory access with a bus of width **w**.

Two memory accesses *u* and *v* in *G'=EDG(G,S,σ)* are compactable if:

- *u* and *v* are the same kind of operation (load or store) to the same variable.
- there is a stride edge between *u* and *v* with $\sigma_{(u,v)}$=1.
- there is no path in *G* between *u* and *v*. (i.e. the execution of none of them depends on the execution of the other).

Let **w** be the width of the bus. The compaction algorithm is as follows:

- Apply the algorithm to deduce new stride edges to the graph *G'=EDG(G,S,σ)*.
- Remove all stride edges with $\sigma_{(u,v)}\neq 1$ from *G'*.
- Mark compactable accesses chains assuming a bus with infinite width.
- Divide the compactable accesses chains in blocks of **w** consecutive elements one following the other (the last block might have less than **w** elements).

To test the effectiveness of the compaction algorithm, we have compared the results obtained by this algorithm with the assembly code generated by the AIX Fortran77 compiler version 3.2 of an IBM SP2 machine, based on POWER2 processors (that have wide buses). In all tests performed, both, the AIX Fortran77 compiler and our algorithm were able to compact memory operations that have stride 1 in the original loop (like in the example of Figure 2). However, in situations like the one in Figure 4, the AIX Fortran77 compiler failed to compact the two memory operations (for instance B^1 and A^0). On the contrary, the algorithm proposed, can deduce new stride edges, which gives more opportunities to compact memory accesses. For the evaluations performed in this paper it must be noticed that in none of the tests performed, our algorithm produced worse results than the AIX Fortran77 compiler and, in the loops tested (that we will describe in the next section) 492 new stride edges were deduced, 150 of them with stride 1.

5. EVALUATION OF WIDE BUSES

5.1 Workbench

The workbench is composed of 1180 loops of the Perfect Club [BCKK88], which account for 77.82% of the total execution time (on an HP 9000/735 workstation). The extended dependence graphs of all the innermost DO loops composed of a single basic block have been obtained with the ICTINEO compiler [ABG+96]. 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 block loops using IF-conversion [AKW83].

Program	% Execution time	% Memory operations	% Mem. ops. with stride 1	% Loops with recurrences
ADM	78.73	38.31	71.04	48.43
QCD	42.81	54.30	86.09	23.64
MDG	62.33	46.72	67.25	14.39
TRACK	30.0	62.58	75.85	36.00
BDNA	68.89	18.92	45.25	98.83
OCEAN	96.79	43.67	26.10	72.24
DYFESM	97.5	58.47	89.49	58.43
MG3D	69.81	20.20	52.56	33.54
ARC2D	94.77	40.24	59.77	1.28
FLO52	92.29	42.30	90.73	1.69
TRFD	97.22	60.21	66.31	6.93
SPEC77	84.66	33.84	35.63	33.69

Table 1: Some characteristics of the Perfect Club Loops

Table 1 shows some characteristics of the loops studied. The first column shows, for each program, the percentage of the total execution time spent in the loops selected. The second column shows the (dynamic) percentage of operations in the loops that are memory operations. The third column shows the percentage of these memory operations that have stride 1 (with an unroll factor of two and after applying the algorithm to deduce stride 1 edges). Finally, the fourth column shows the percentage of loops that have recurrences. Recurrences can override the potential performance benefits of wide buses mainly because recurrence circuits can transform memory-bound loops (in terms of percentage of memory operations) to compute-bound loops (in terms of the number of cycles required per iteration), so the additional memory bandwidth may not have any effect on performance. In addition, recurrence circuits increase the likelihood of paths between compactable memory accesses, preventing their compaction.

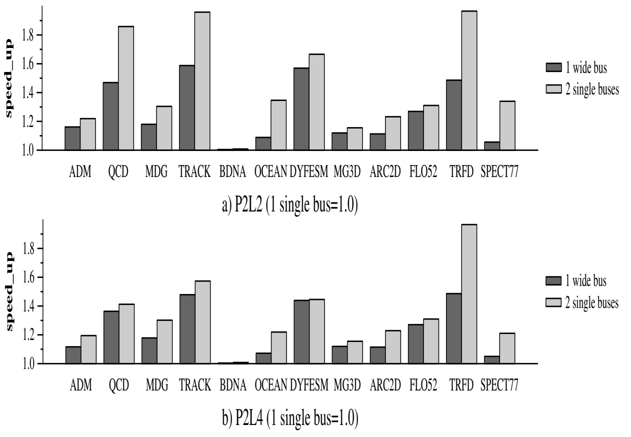

Figure 5: Performance for different bus configurations

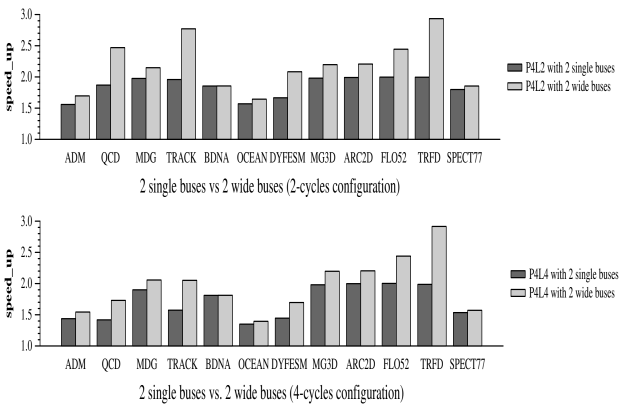

Figure 6: 2 single buses vs. 2 wide buses. (baseline= 1 single bus with a P2L2 configuration in Figure a, and 1 single bus with a P2L4 configuration in Figure b).

In order to schedule the loops, HRMS has been used; it is a scheduling technique that obtains near optimal results in terms of *II*, while reducing the register requirements. HRMS achieves optimal execution time (*II=MII*) for 97.5% of the loops tested

Configuration	Characteristics	Type of functional unit		
		Adder	Multiplier	Div/SQRT
P2L2	#FUs	2	2	1
	Latency	2	2	16/30
	Pipelined	yes	yes	not
P2L4	#FUs	2	2	1
	Latency	4	4	16/30
	Pipelined	yes	yes	not
P4L2	#FUs	4	4	2
	Latency	2	2	16/30
	Pipelined	yes	yes	not
P4L4	#FUs	4	4	2
	Latency	4	4	16/30
	Pipelined	yes	yes	not

Table 2: Processor core configurations

5.2 Performance evaluation

In order to evaluate the performance of different bus configurations, we have generated the software pipelined schedule for all the loops using several target configurations. For simplicity, we assume a set of VLIW-like processor core configurations, which are summarized in Table 2. Nevertheless, in Section 6 we evaluate wide buses for statically scheduled superscalar processors.

We labelled the different processor core configurations by *PxLy*, having the configuration *x* adders, *x* multipliers and *x/2* dividers and being *y* the latency of the adder and the multiplier. All functional units are fully pipelined except the divider which is not pipelined at all. With these processor core configurations we test several bus configurations (shown in Figure 1). In all cases loads have two cycle latency and are fully pipelined, and stores have one cycle latency.

Figure 5a shows the speed-up[1] of 1 wide bus and 2 single buses over 1 single bus for a the processor core configuration P2L2. The speed-ups obtained increasing the memory bandwidth vary significantly according to the characteristics of the program (see Table 1). The programs with better speed-ups when two buses are provided (QCD, TRACK, DYFESM and TRFD) are programs with a high percentage (more than 50%) of memory operations (i.e. memory bound programs). On the contrary, compute bound programs like BDNA (19% of memory operations) show a negligible speed-up.

Notice that, in general, wide buses provide noticeable speed-ups (up to 58.75% faster in the TRACK program) relative to a single bus. However, the performance of a wide bus is always below that of two buses. Some of the programs (the most significant cases are DYFESM and FLO52) perform almost as well with a wide bus than with two buses. Notice that these two programs have a high percentage (89% and 90% respectively) of memory operations with stride 1. On the contrary, the programs for which a wide bus performs noticeable worse than two buses (for instance OCEAN and SPEC77) have a limited percentage of stride 1 accesses (26% and 35% respectively).

Figure 5b shows the speed-up of the same bus configurations for the P2L4 configuration. Notice that, in general, programs exhibit less performance improvement when extra bandwidth is added than in the P2L2 configuration. This is because increasing the latency of operations increases the length of recurrence circuits, that can convert loops with recurrences from memory-bound to compute-bound. The programs that show almost the same speed-up than in Figure 5a (ARC2D, FLO52 and TRFD) have practically no loops with recurrences. Also, because programs are more compute bound they use a smaller fraction of the bandwidth available (yet, they still require more than one bus). For this reason in some programs (QCD, TRACK and DYFESM) the wide bus configuration performs noticeable closer to the two bus configuration.

In the evaluations performed, we have used processor configurations very similar to the existent in most advanced microprocessors. However, future designs may have more functional units and stress the memory subsystem to maximum. In order to provide future microprocessors with the required memory bandwidth, it is more feasible (in terms of cost) to widen the buses than increasing their number. Since there are already microprocessors with two buses, it will have little additional cost to implement processors with two wide buses[2], while building microprocessors with four single buses will be noticeable costly. Figure 6 shows the improvement that two wide buses provide over two single buses. For this evaluation we have considered more aggressive configurations (P4L2 and P4L4) that are closer to expected future designs. Since the bandwidth of the baseline configurations (one single bus) is insufficient for the aggressive configurations we are using, pro-

1.The execution time of each loop has been estimated as the number of iterations times the *II* found by HRMS.

2.The IBM POWER2 [WhDh94] has been already implemented with two wide buses.

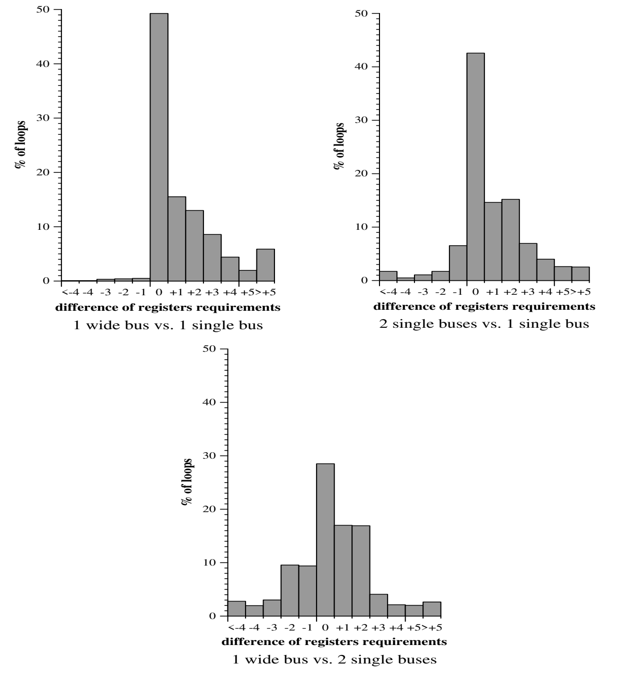

Figure 7: Study of additional register requirements

grams show speed-ups closer to two when two wide buses are considered. Notice that in programs with a percentage of memory operations over 50% (QCD, TRACK, DYFESM and TRFD), widening the buses provides a noticeable speed-up (up to 47%), specially for configuration P4L2. In general, converting two buses in two wide buses achieves a good increment of performance. The geometrical mean of relative speed-ups for both processors core configurations (P4L2 and P4L4) is 14.17% and 12.17% respectively. However, the geometrical mean for the most bandwidth demanding programs (QCD, TRACK, DYFESM and TRFD) is 35.41% for P4L2 and 30.73% for P4L4. Also, as reported previously, programs whose loops have a high number of recurrence circuits, benefit less from the additional bandwidth, when the latency of functional units is increased.

5.3 Register requirements

Adding more buses to a processor can increase the register requirements of a loop mainly because a more aggressive bus configuration can reduce the *II* of the scheduled loop. Register pressure is, to some extent, proportional to the number of concurrently executed iterations. Reducing the *II*, in general, increases the number of stages in which the schedule is divided, and therefore, the number of simultaneously overlapping iterations. In addition, the lifetimes can be larger because an smaller *II* imposes more resource constrains than a larger one. In the case of wide buses, in addition, the lifetime of a value can be increased because the other value of the wide operation is not ready yet to be stored or must be loaded a few cycles before.

If the additional register requirements of wide buses are high, the performance advantage can be counteracted by the performance lost due to additional spill code. Figure 7 shows a comparison of additional register requirements of the loops for the bus configurations of Figure 1 using the P2L2 configuration. The horizontal axis show the difference in terms of registers required for three bus configurations, and the vertical axis represents the percentage of loops that differs in an specific number of registers.

Figure 7a compares one wide bus versus one single bus. In this case, 49.3% of the loops require the same amount of registers in both cases, 15.5% of the loops require one more register in a wide bus configuration than in a single bus configuration, and so on. Notice that, even though a wide bus requires almost always the same number or more registers, the additional number of registers is small (only 5.8% of the loops require more than 5 additional registers).

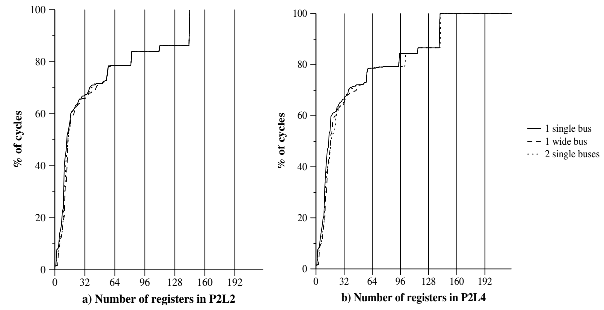

Figure 8: Dynamic cumulative distribution of register requirements.

Figure 7b, shows the additional register requirements of two single buses versus one single bus. Notice that the behaviour is very similar to Figure 7a. This suggests that the additional register requirements are mainly due to the smaller *II*s that can be obtained because of the extra bandwidth.

Figure 7c compares one wide bus versus two single buses. In this case only 28.5% of the loops have the same register requirements. A total of 26.5% of the loops require less registers with a wide bus configuration than with two buses, because of the constraints that avoid a wide bus to achieve the same performance levels than two single buses. Finally 45% of loops have more register requirements with one wide bus than with a two buses configuration. Notice, however, that most of the loops that require additional registers, require only one or two more registers.

A detailed analysis of the loops reveals that the ones that have an important difference in registers requirements are those that use a small number of registers. In general, the loops with a high number of register requirements have minor differences, specially when considering execution time. Figure 8 compares the dynamic register requirements of the different bus configurations. This figure shows the percentage of execution time spent by loops that can be scheduled with a given number of registers. For instance, in Figure 8a the loops that spent near 80% of execution time in P2L2 configuration can be scheduled with a 64-register file in all the bus configurations. The loops that require more than 64 registers have the same behaviour for the different bus configurations. The same behaviour than in P2L2 can be observed in the dynamic register requirements of P2L4 (Figure 8b) where there are more registers required because the latency of the functional units is greater than in P2L2.

In conclusion, the increase of register requirements of a wide bus configuration does not have a significant impact on the final performance since the main (relative) differences are in loops with low register requirements (that can be scheduled without needs of spill code).

6. WIDE BUSES ON SUPERSCALAR PROCESSORS

Up to now, we have evaluated wide buses considering VLIW-like processor models; however, most commercial microprocessors are superscalar processors[SmSo95]. Superscalar processors have some architectural characteristics that might difficult the addition of wide buses.

Because hardware and software evolve, most processor designs inherit a legacy from their predecessors. One of the main compo-

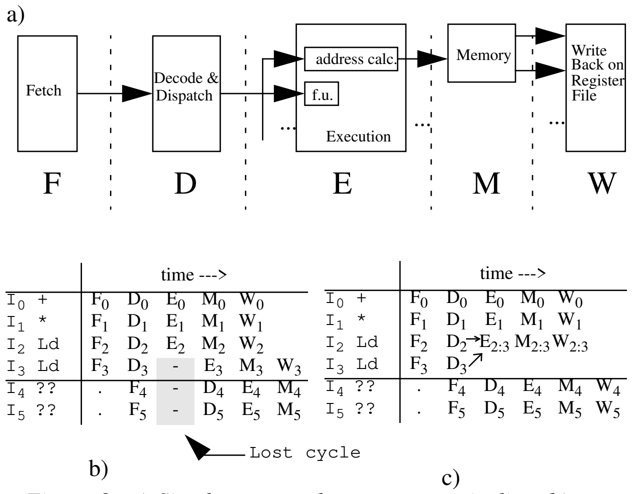

Figure 9: a) Simple superscalar processor pipeline. b) Execution of instructions without compaction hardware. c) Execution of instructions with the hardware proposed.

nents of this legacy is binary compatibility; a superscalar processor must be able to execute a machine program written for an earlier generation processor. If wide buses are added to a superscalar processor, the instruction set must change, because wide-instructions are needed to benefit from the additional bandwidth. Including *load wide* and *store wide* instructions in the instruction set might change the number of bits on the operation code field, because more instructions have been added. Changing the instruction set can break off binary compatibility and, in particular, backward compatibility (i.e. code compiled for a new generation can not be executed on previous generations). For instance, the POWER2 has wide load/store instructions (load quadword and store quadword), and can execute code compiled for its predecessor, the RS/6000. However, code compiled for a POWER2 to take advantage of wide buses cannot be executed on a RS/6000.

In [WOR96], several techniques were proposed to exploit wide buses without changing the instruction set. These techniques are based on queuing loads and moving to the processor the cache line that contains the required data; pending loads that use data in this cache line are also served. These techniques are applied during the memory phase and they are only applicable to dynamically scheduled processors. Moreover, they still require the generation and translation of multiple addresses per cycle. In this section we propose a hardware mechanism to detect compactable memory accesses during the decode/dispatch phase of execution that can be applied to both statically and dynamically scheduled superscalar processors, and only requires one address translation per wide access.

In a superscalar processor the number of memory operations that can be issued at the same cycle is limited by the number of memory units available. Figure 9a, shows a simple pipeline for a basic superscalar processor. As an example, consider a statically scheduled four issue superscalar processor (similar in some aspects to the Alpha 21164 [ERPR95]) but with a single memory unit. If two load operations are fetched in the same cycle (as I_2 and I_3 in Figure 9b), the load I_3 cannot be issued, stalling the operations that follow it for one cycle. Adding a simple hardware to the dispatcher may allow the processor to compact two compactable memory accesses like I_2 and I_3 and issue only one wide operation ($I_{2:3}$ in Figure 9c) that performs both load operations simultaneously allowing the following instructions to proceed without any delay. With this proposal two compactable memory references still require two issue slots, however they only require one memory unit (with one TLB port, a single ported cache, etc...).

The mostly used addressing mode is *base+displacement,* which in some RISC processors is the only available addressing mode. If two consecutive loads (stores) have the same base register (usually coded with 5 bits) and the displacements (usually 16 bits) differ by *operand size* (usually 4-byte or 8-byte long) then both operations can be issued at the same cycle using only one memory resource (as in Figure 9c). Figure 10 shows the hardware to detect pairs of compactable memory operations. To benefit from this hardware, accesses that can be compacted must be scheduled consecutively in the same instruction packet (as I_2 and I_3 are in the instruction packet I_0 I_1 I_2 I_3). The use of the compacting method described in previous sections allows the compiler to schedule these memory accesses together.

In order to evaluate this proposal, two machine models are compared: a machine with wide instructions (**MW**) where the instruction set has been extended to support wide-memory instructions, and a machine without wide-memory instructions but with wide buses and the described hardware support (**MH**). For each machine model, two processor core configurations have been evaluated: a 4-issue processor with the P2L2 functional unit configuration and one wide bus; and a 8-issue processor with the P4L2 functional unit configuration and two wide buses.

The speed-ups of the 4-issue MW model over a single bus 4-issue machine are very similar to the P2L2 VLIW-like processor of Section 5 (up to 57.30% for the TRACK program), despite the additional constraint of having only 4 issue slots. Something similar occurs for the 8-issue machine, which however, shows slightly lower speed-ups due to the more aggressive configuration of the base machine (8-issue with 2 single buses).

Despite that a wide load requires two issue slots in the MH model and only one in the MW model, the performance of MH is very similar to MW[1]. On the average, MH achieves 99.89% of the speed-up achieved by MW for a 4-issue machine, and 97.74% of the speed-up of MW for a 8-issue machine. Therefore, using the mechanism proposed, statically scheduled superscalar processors can have the performance advantages of wide buses without losing binary compatibility.

In addition, even though most of the recently announced microprocessors are 64-bit, they provide support for 32-bit data manipulation (there are programs that only use `real*4` variables). So, even with only a single (64-bit wide) bus, they can use (in future implementations) this mechanism to speed-up 32 bit data manipulation by compacting two 32-bit memory accesses into a single 64-bit access.

7. CONCLUSIONS

High performance microprocessors have high bandwidth demands between the register file and the first level of the memory hierarchy. Two alternatives have been evaluated in this paper: increasing the number of memory ports or widen the ports available. Dou-

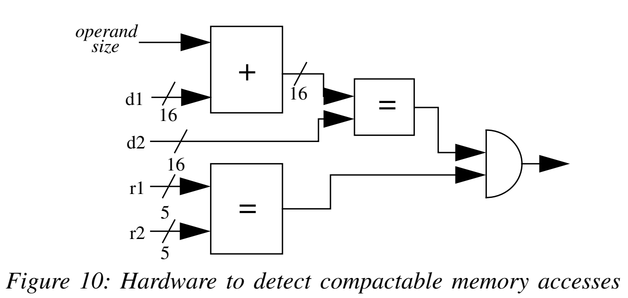

Figure 10: Hardware to detect compactable memory accesses

1.We do not show detailed results since the speed-ups of the MH model are practically indistinguishable from the speed-up of the MW model.

bling the width of the bus provide the same *peak* bandwidth than doubling the number of buses at a fraction of the cost (area and access time). However, to effectively exploit wide buses (specially with statically scheduled machines) requires especial compilation techniques and has some drawbacks such as lost of binary compatibility.

In this paper we have presented a compilation technique to pack two stride-one memory accesses into a unique wide memory access. Although vectorizing techniques can be used to detect wide accesses (a wide access can be seen as a stride-one vector memory access with vector length two), the technique presented offers some additional benefits since it can compact independent memory operations that access consecutive locations, as well as consecutive accesses to vectors.

We have evaluated the performance improvement of wide buses over single buses. Having a wide bus shows speed-ups up to 58% over a single bus. However, since exploiting it requires the existence (and the detection at compile time) of stride-one memory accesses, a wide bus offers less performance improvements than two single buses. The performance of a wide bus (in a configuration with 2-cycle latency functional units) was between 75.6% and 99.6% of the performance of two buses; however, for 7 of the 12 programs tested it was over 90% of two buses. In 4-cycle latency configurations, programs become less memory bound reducing the gap between a wide bus and two buses (9 of 12 programs obtained a performance over 90% of two buses with a wide bus).

Wide buses impose more constraints to the scheduler, which can increase the register requirements. If the additional register requirements are high, the performance advantage of wide buses can be counteracted by the performance lost due to additional spill code. However, the evaluation performed reveals that the additional register requirements (and specially their effects) are insignificant.

Finally we have proposed a hardware mechanism that tries to compact two load/stores into a single wide load/store at run time, so that only a single wide bus is used. Even though this alternative wastes two issue slots per wide load performed, it produces almost the same performance improvement than having wide load/store instructions (97.7% of the performance of a machine with wide load/stores in the worst case). This hardware technique permits to exploit the benefits of wide buses for statically scheduled processors, while preserving binary compatibility.

ACKNOWLEDGEMENTS

The authors would like to thank the rest of the ICTÍNEO team for their support to this work. This work has been supported by the Ministry of Education of Spain under contract TIC 429/95. We also acknowledge C^4 (Centre de Computació i Comunicacions de Catalunya) for providing access to the IBM SP2 machine.

REFERENCES

[ABG+96] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero. *Ictineo: a Tool for Instruction Level Parallelism Research*. RR UPC-DAC-1996-61. Dec. 1996.

[AKW83] J.R. Allen, K. Kennedy and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th annual Sym on Principles of Programming Languages*, January 1983.

[BCKK88] M. Berry, D. Chen, P. Koss and D. Kuck. *The Perfect Club benchmarks: Effective performance evaluation of supercomputers*. Tech. Rep. 827, C. for Supercomputing Res. and Dev.., Univ. of Illinois at Urbana-Champaign, Nov. 1988.

[CLC+95] 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.

[DeTo93] J.C. Dehnert and R.A. Towle. Compiling for cydra 5. *Journal of Supercomputing*, 7(1/2):181-227, 1993.

[DoHi79] J. Dongarra and A.R. Hind. Unrolling loops in FORTRAN. *Soft.-Practice and Exp.*, 9, 3(Mar.), 219-226, 1979.

[ERPR95] 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.

[GAB+88] R.B. Garner, A. Agrawal, F. Briggs, E.W. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnik, M. Namjoo, D. Patterson, J. Pendleton and R. Tuck. The scalable processor architecture (SPARC). *CompCon 88, pp.* 278-283, 1988.

[Hsu94] P.Y.T. Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23-33, April 1994.

[Huf93] R.A. Huff. Lifetime-sensitive modulo scheduling. In *6th Conference on Programming Language, Design and Implementation*, pp. 258-267, 1993

[Hun95] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *CompCon 95*, 1995.

[IBM94] IBM Inc. *RISC System/6000 PowerPC System Architecture*. Ed. by Frank Levine and Steve Thurber. Morgan Kaufmann Publishers Inc. San Francisco, Cal. 1st ed July 1994.

[JoAl90] R.B. Jones and V.H. Allan. Software pipelining: A comparison and improvement. *MICRO-23*, pp.46-56,Nov. 1990.

[Jol91] R. Jolly. A 9-ns 1.4 gigabyte 17-ported CMOS register file. IEEE J. of Solid-State Circuits,25:1407-1412,Oct. 1991.

[LaHw95] D.M. Lavery and W-M.W. Hwu. *Unrolling-Based Optimizations for Modulo Scheduling*. MICRO-28, pp. 327-337. Dec. 1995.

[LMM92] R.B. Lee, M. Mahon, and D. Morris. Pathlenght reduction features in the PA-RISC architecture. In *CompCon 92*, pp.129-135, 1992.

[LVAG95] J. Llosa, M. Valero, E. Ayguadé and A. González. *Hypernode Reduction Modulo Scheduling*. MICRO-28, pp. 350-360. Dec. 1995.

[LVAL94] J. Llosa, M. Valero, E. Ayguadé and J. Labarta. Register requirements of pipelined loops and their effect on performance. In *Proc. 2nd Int Workshop on Massive Parallelism: Hardware, Software and Applications*, pp 173-189 October 1994.

[MAD92] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. Register requiremets of pipelined processors. In *Int. Conf. on Supercomputing*, pp. 260-271, July 1992.

[RaGl81] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Ann. Microprogramming Workshop*, pp. 183-197, Oct. 1981.

[Rau94] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. MICRO-27,pp. 63-74, Nov. 1994.

[RLTS92] B.R. Rau, M. Lee, P. Tirumalai and P. Schlansker. Register allocation for software pipelind loops. In *Proc. of the Conf. on Prog. Lang. Design and Impl.*, pp. 183-199, June 1992.

[SmSo95] J. E. Smith and G.r S. Sohi. The Microarchitecture of Superscalar Processors. In *Proceedings of the IEEE*, Dec. 1995.

[SoFr91] G. S. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. ASPLOS-IV, pp. 53-62, April 1991.

[WeEs88] N. Weste and K. Eshraghian. Principles of CMOS VLSI Design: A systems Perspective. Addison-Wesley Pub. 1988.

[WhDh94] S.W.White and S. Dhawan. *POWER2: Next Generation of the RISC System/6000 family.* IBM J. Res. Develop. 38, No. 1, 493-502 (September 1994)

[WOR96] K.M. Wilson, K. Olukotun, and M. Roseblum. Increasing cache port efficiency for dynamic superscalar microprocessors. ISCA-23, May 1996.