Speculative Dynamic Vectorization

Alex Pajuelo, Antonio González and Mateo Valero
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona – Spain
{mpajuelo, antonio, mateo}@ac.upc.es

Abstract

Traditional vector architectures have shown to be very effective for regular codes where the compiler can detect data-level parallelism. However, this SIMD parallelism is also present in irregular or pointer-rich codes, for which the compiler is quite limited to discover it. In this paper we propose a microarchitecture extension in order to exploit SIMD parallelism in a speculative way. The idea is to predict when certain operations are likely to be vectorizable, based on some previous history information. In this case, these scalar instructions are executed in a vector mode. These vector instructions operate on several elements (vector operands) that are anticipated to be their input operands and produce a number of outputs that are stored on a vector register in order to be used by further instructions. Verification of the correctness of the applied vectorization eventually changes the status of a given vector element from speculative to non-speculative, or alternatively, generates a recovery action in case of misspeculation.

The proposed microarchitecture extension applied to a 4-way issue superscalar processor with one wide bus is 19% faster than the same processor with 4 scalar buses to L1 data cache. This speed up is due basically to 1) the reduction in number of memory accesses, 15% for SpecInt and 20% for SpecFP, 2) the transformation of scalar arithmetic instructions into their vector counterpart, 28% for SpecInt and 23% for SpecFP, and 3) the exploitation of control independence for mispredicted branches.

1. Introduction

Vector processors [2, 6, 10, 16] are very effective to exploit SIMD parallelism, which is present in numerical and multimedia applications. Vector instructions' efficiency comes from streamed memory accesses and streamed arithmetic operations. Traditionally, the burden to exploit this type of parallelism has been put on the compiler and/or the programmer [1, 24].

The average programmer can deal with codes whose vectorization is relatively straightforward. The compiler has a partial knowledge of the program (i.e. it has a limited knowledge of the values of the variables). Because of that, it generates code that is safe for any possible scenario according to its knowledge, and thus, it may loose significant opportunities to exploit SIMD parallelism. On top of that, we have the problem of legacy codes that have been compiled for former versions of the ISA with no SIMD extensions and therefore are not able to exploit new SIMD extensions incorporated in newer ISA versions.

In this paper we present a mechanism for dynamically generating SIMD instructions (also referred to as vector instructions). These SIMD instructions speculatively fetch and precompute data using the vector units of the processor. This vectorization scheme works even for codes in which a typical vectorizing compiler would fail to find SIMD parallelism. The vectorization process begins when the processor identifies a load operation that is likely to have a constant stride (based on its past history). Any instruction whose operands have been vectorized is also vectorized. In this way, the 'vectorizable' attribute is propagated down the dependence graph. The scalar unit has to verify the correctness of the vectorization. For this purpose, the corresponding scalar instructions are converted into 'check' operations that basically validate that the operands used by the vector instructions are correct. This is very similar in concept to other checker proposed in the past for other purposes such as fault tolerance [3, 13]. In case of a vectorization misspeculation, the current instances of the vector instructions are executed in scalar mode.

The proposed mechanism also allows the processor to exploit control-flow independence. When a branch is mispredicted, the scalar pipeline is flushed but the vector operations are not squashed. The new scalar instructions corresponding to the correct path will check whether the operations performed by the vector instructions are still correct and if this is the case, their results will be committed. Otherwise, a vectorization misspeculation will be fired, which will discard the vector results.

With the dynamic vectorization mechanism the number of memory requests decreases by 15% for SpecInt and 20% for SpecFP, and the number of executed arithmetic instructions is 25% lower for SpecInt in a 4-way superscalar processor with one wide bus. This reduction in memory requests and executed instructions results in significant speedups. For instance, a 4-way superscalar processor with dynamic vectorization and one wide data cache bus is 19% faster than a 4-way superscalar processor with 4 scalar buses and no dynamic vectorization.

The rest of the paper is organized as follows. Section 2 motivates this work by presenting some statistics about strided memory accesses. Section 3 presents the hardware implementation of the dynamic vectorization mechanism. Performance statistics are discussed in section 4. Section 5 outlines the related work. We conclude in section 6.

2. Motivation

Strided memory loads [7,8] are the instructions that fire the proposed speculative dynamic vectorization mechanism. To identify a strided load, at least three dynamic instances of the static load are needed. The first dynamic instance sets the first memory address that is accessed. The second dynamic instance computes the initial stride, subtracting the memory address of the first dynamic instance from the current address. The third dynamic instance checks if the stride is repeated computing the current stride and comparing it with the first computed stride.

Figure 1 shows the stride distribution for SpecInt95 and SpecFP95 (for this figure, the stride is computed dividing the difference of memory addresses by the size of the accessed data). Details of the evaluation framework can be found in section 4.1.

As shown in Figure 1, the most frequent stride for SpecInt95 and SpecFP95 is 0. This means that dynamic instances of the same static load access the same memory address. For SpecInt this stride is due, mainly, to the accesses of local variables and memory addresses referenced through pointers. For SpecFP the stride 0 is mainly due to spill code.

Usually, for SpecFP, the most frequent stride is stride 1 because these applications execute the same operations over every element of some array structures. However, due to the code optimizations [4,9] included by the scalar compiler, such as loop unrolling, some stride 1 accesses become stride 2, 4 or 8. The bottom line of this statistics is that strided accesses are quite common both in integer and FP applications.

The results in Figure 1 also suggest that a wide bus to the L1 data cache can be very effective at reducing the number of memory requests. For instance, if the cache line size is 4 elements, multiple accesses with stride lower than 4 can be served with a single request if the bus width is equal to the line size. These types of strides represent 97,9% and 81,3% of the total strided loads for SpecInt95 and SpecFP95 respectively.

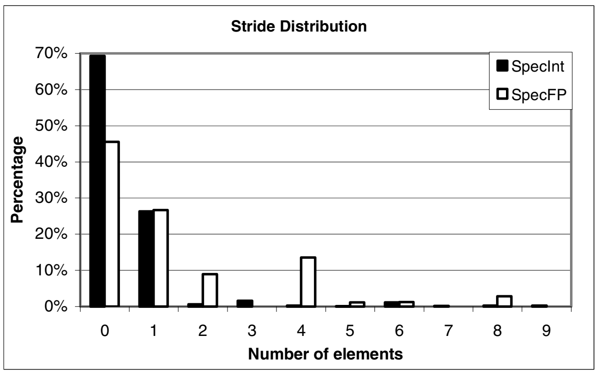


***Figure 1**. Stride distribution for SpecInt95 and SpecFP95.*

3. Dynamic vectorization

The microarchitecture proposed in this work is a superscalar core extended with a vector register file and some vector functional units. Figure 2 shows the block diagram of the processor where black boxes are the additional vector resources and related structures not usually found in a superscalar processor, and grey boxes are the modified structures to implement the speculative dynamic vectorization mechanism.

3.1. Overview

Speculative dynamic vectorization begins when a strided load is detected. When this happens, a vectorized instance of the instruction is created and it is executed in a vector functional unit storing the results in a vector register. Next instances of the same static instruction are not executed but they just validate if the corresponding speculatively loaded element is valid. This basically consists in checking that the predicted address is correct and the loaded element has not been invalidated by a succeeding store. Every new instance of the scalar load instruction validates one element of the corresponding destination vector register.

Arithmetic instructions are vectorized when any of the source operands is a vector register. Succeeding dynamic instances of this instruction just check that the corresponding source operands are still valid vector elements (details on how the state of each element is kept is later explained).

When a validation fails, the current and following instances of the corresponding instruction are executed in scalar mode, until the vectorizing engine detects again a new vectorizable pattern. With this dynamic vectorization mechanism, as shown in Figure 3, with unbounded resources, 47% of the SpecInt95 instructions and 51% of the SpecFP95 instructions can be vectorized.

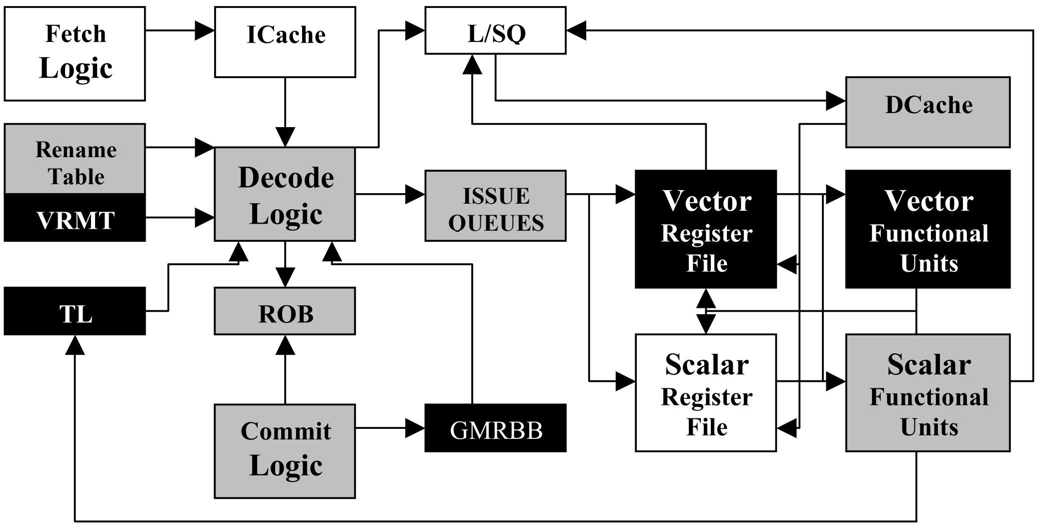


***Figure 2**.Block diagram of a superscalar processor with the speculative dynamic vectorization mechanism. Black blocks are structures or resources added and gray blocks are structures modified.*

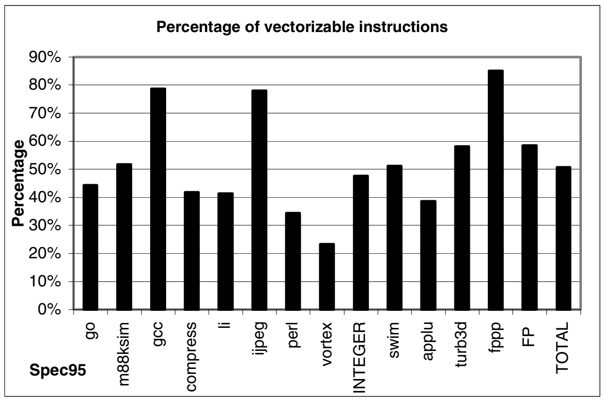


***Figure 3**: Percentage of vectorizable instructions.*

3.2. Instruction vectorization

The first step to create vector instances of an instruction is detecting a strided load [7,8]. To do this, it is necessary to know the recent history of memory accesses for every load instruction. To store this history the processor includes a Table of Loads (TL in Figure 2) where, for every load, the PC, the current address, the stride and a confidence counter are stored as shown in Figure 4.

PC	Last Address	Stride	Confidence Counter

***Figure 4**.Table of Loads.*

When a load instruction is decoded, it looks for its PC in the TL. If the PC is not in this table, the last address field is initialized with the current address of the load and the stride and confidence counter fields are set to 0.

Next dynamic instances compute the new stride and compare the result with the stride stored in the table, increasing the confidence counter when both strides are equal or resetting it to 0 otherwise. When the confidence counter is 2 or higher, a new vectorized instance of the instruction is generated. The last address field is always modified with the current memory address of the dynamic instance.

When a vectorized instance of an instruction is generated, the processor allocates a vector register to store its result. The processor maintains the associations of vector registers and vector instructions in the Vector Register Map Table (VRMT in Figure 2). This table contains, for every vector register the PC of the associated instruction, the vector element (offset) corresponding to the last fetched scalar instruction that will validate (or has validated) an element of this vector, the source operands of the associated instruction, and, if the instruction is vectorized with one scalar operand and one vector operand, the value of the scalar register is also stored, as shown in Figure 5.

PC	Offset	Source Operand 1	Source Operand 2	Value

***Figure 5**.Contents of each entry of the Vector Register Map Table.*

Every time a scalar instruction is fetched, this table is checked and if its PC is found the instruction is turned into a validation operation. In this case, the offset field determines which vector element must be validated and then, the offset is incremented. In the case that the offset is equal to the vector register length, another vectorized version of the instruction is generated and a free vector register is allocated to it. The VRMT table entry corresponding to this new vector register is initialized with the content of the entry corresponding to the previous instance, excepting the field offset, which is set to 0.

The register rename table is also modified (see Figure 6) to reflect the two kind of physical registers (scalar and vector). Every logical register is mapped to either a physical scalar register or a physical vector register, depending on whether the last instruction that used this register as destination was vectorized or not. Every entry of the table contains a V/S flag to mark if the physical register is a scalar or a vector register and the field offset indicates the latest element for which a validation has entered in the pipeline.

Physical Register	V/S	Offset

Figure 6.*Entry of the modified Rename Table.*

When every instruction is decoded the V/S flags (vector/scalar) of their source operands are read and if any of the two is set to V, the instruction is vectorized. In parallel, the VRMT table is accessed to check if the instruction was already vectorized in a previous dynamic instance. If so, the instruction is turned into a validation operation. Validation is performed by checking if the source operands in the VRMT table and those in the rename table are the same. If they differ, a new vectorized version of the instruction is generated. Otherwise, the current element pointed by offset is validated and this validation is dispatched to the reorder buffer in order to be later committed (see next section for further explanation). Besides, if the validated element is the last one of the vector, a new instance of the vectorized instruction is dispatched to the vector data-path.

An arithmetic instruction that has been vectorized with one vector source operand and one scalar register operand, waits in the decode stage, blocking the next instructions, until the value of the physical register associated to the scalar source operand is available. Then, it checks if the value of the register matches the value found in the VRMT and if so, a validation is dispatched to the reorder buffer. Otherwise, a new vectorized instance of the instruction is created. This stalls do not impact much performance since the number of vectorized instructions with one scalar operand that is not ready at decode is low. Figure 7 shows the differents IPC's obtained blocking these instructions (black bar) and the ideal case (white bar) where no one of these instructions are blocked.

Note that the cost of a context switch is not increased since only the scalar state of the processor needs to be saved. The additional structures for vectorization are just invalidated on a context switch. When the process restarts again the vectorization of the code starts from scratch at the point where the process was interrupted.

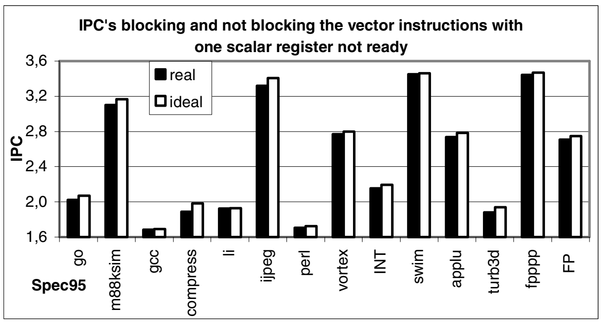


Figure *7.IPCs obtained blocking (real) and not blocking (ideal) the vector instructions with one scalar register not ready for a 4-way processor with 1 port and 128 vector registers.*

3.3. Vector registers

Vector register is one of the most critical resources in the processor because they determine the number of scalar instructions that can be vectorized. Vector registers can be regarded as a set of scalar registers grouped with the same name.

A vector register is assigned to an instruction in the decode stage when this instruction is vectorized. If no free vector register is available, the instruction is not vectorized, and continues executing in a scalar mode.

To manage the allocation/deallocation of vector registers, each register contains a global tag and each element includes a set of flags of bits as shown in Figure 8.

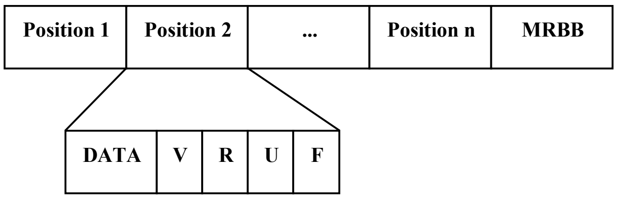


Figure *8.(Top) Vector register structure for dynamic vectorization.*

The V (Valid) flag indicates whether the element holds committed data. This bit is set to 1 when the validation associated to the corresponding scalar instruction commits.

The R (Ready) flag indicates whether the element has been computed. Depending on the kind of instruction the data will be ready when is brought from memory or computed by a vector functional unit.

When a validation of an element has been dispatched but not committed yet, the U (Used) flag is set to 1. This prevents the freeing of the physical register until the validation is committed (details on the conditions to free a vector register are described below).

The F (Free) flag indicates whether the element is not longer needed. This flag is set to 1 when the next scalar instruction having the same logical destination register or its corresponding validation commits.

A vector register will be release when all its computed elements have been freed (i.e. are not needed any more). Besides, a register is also released if all validated elements are freed and no more elements need to be produced. In order to estimate when no more elements will be computed, we assume that this will happen when the current loop is terminated. For this purpose, the processor includes a register that is referred to as GMRBB (Global Most Recent Backward Branch) that holds the PC of the last backward branch that has been committed [19]. Each vector register stores in the MRBB (Most Recent Backward Branch) tag the PC of the most recently committed backward branch when the vector register was allocated. This backward branch, usually, coincides with the last branch of a loop, associating a vector register to an instruction during some iterations.

A vector register is freed when one of the following two conditions holds:

1) All vector elements have the flags R and F set to 1. This means that all elements have been computed and freed by scalar instructions.

2) Every element with the flags V set, has the flag F set, and all the elements have the flag R set and flag U cleared, and the content of the tag MRBB is different of the register GMRBB. This means that all the validated elements have been freed. Furthermore, all elements have been computed and no element is in use by a validation instruction. It is very likely that the loop where the vector operation that allocated the register was, has been terminated.

3.4. Vector data path

Vector instructions wait in the vector instruction queues until their operands are ready and a vector functional unit is available (i.e. instruction are issued out-of-order). Vector functional units are pipelined and hence can begin the computation of a new vector element every cycle. Every time an element is calculated, the vector functional unit sets to 1 the flag R associated to that position, allowing others functional units to use it.

Vector functional units can compute operations having one vector operand and one scalar operand. To do this, the functional units must have access to the scalar register file and the vector register file. In the case of the scalar register, the element is read just once.

Note that some vector instruction can be executed having a different initial offset for their source vector operands. This can happen, for example, when two load instructions begin vectorization in different iterations and their destination vector registers are source operands of an arithmetic instruction. To deal with these cases, vector functional units compare these offsets to obtain the greatest. The difference between this offset and the vector register length determines the number of elements to compute. Fortunately, the percentage of the vector instructions whose source operands' offsets are different from 0 is very low, as shown in Figure 9.

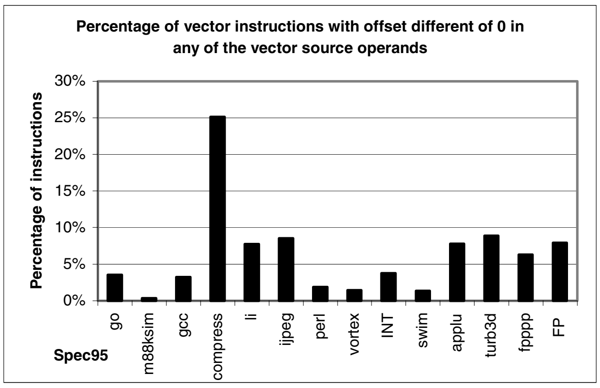


Figure 9.*Percentage of vector instructions with offset different of 0 in any of the vector source operands for an 8-way processor with 128 vector registers.*

3.5. Branch mispredictions and control-flow independence

When a branch misprediction is detected, a superscalar processor recovers the state of the machine by restoring the register map table and squashing the instructions after the branch.

In the proposed microarchitecture, the scalar core works in the same way as a conventional processor, i.e. a precise state is recovered, but vector resources are not modified: vector registers are not freed, and no vector functional unit aborts the execution because they can be computing data that can be used in the future. The objective is to exploit control-flow independence [14, 15, 18]. When the new path enters again in the scalar pipeline, the source operands of each instruction will be checked again, and if it happens that the vector operands are still valid, the instruction does not need to be executed. Figure 10 shows the percentage of instructions in the 100 instructions (100 is a size arbitrarily chosen) that follow a mispredicted branch that do not need to be executed since they were executed in vector mode and continue to have the same source operands after the misprediction.

Note that when a scalar instruction in a wrongly predicted speculative path is vectorized, the vector register may remain allocated until the end of the loop to which the instruction belongs. This wastes vector registers but fortunately only happens for less than 1% of the vector instructions in our benchmarks.

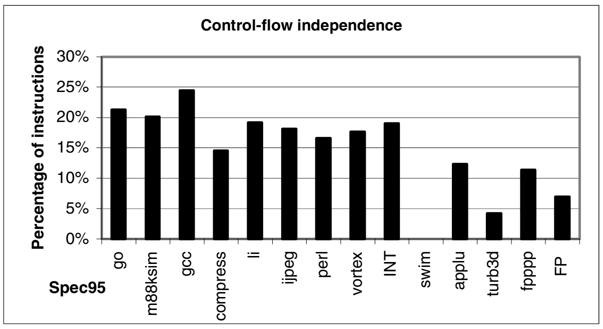


***Figure 10**: Percentage of scalar instructions that are reused after a branch misprediction (limited to the 100 instructions that follow each mispredicted branch).*

3.6. Memory coherence

To ensure memory coherence, stores are critical instructions because these instructions make changes in memory that the processor cannot recover. For this reason, a store instruction modifies the memory hierarchy only when it commits.

A vectorized load copies memory values into a register. However, there may be intervening stores before the scalar load operation that would have made the access in a non-vectorized implementation. Thus, stores must check the data in vector registers to maintain coherence.

For this purpose, every vector register has two fields, the first and the last address of the corresponding memory elements (used only if the associated vector instruction is a load) to indicate the range of memory positions accessed. Stores check whether the addresses that are going to modify are inside the range of addresses of any vector register. If so, the VRMT entry associated to this vector register is invalidated. Then, when the corresponding scalar instruction is decoded, it will not find its PC in the VRMT and another vector instance of this instruction will be created. Besides, all the instructions following the store are squashed.

Fortunately, the percentage of the stores whose memory address is inside the range of addresses of any vector register is low (4,5% for SpecInt and 2,5% for SpecFP).

Due to the complexity of the logic associated to store instructions, only two store instructions can commit in the same cycle.

3.7. Wide bus

To exploit spatial locality a wide bus of 4 words has been implemented. This bus is able to bring a whole cache line every time the cache is accessed. In parallel, the range of addresses held in this cache line are compared with the addresses of pending loads, and all loads that access to the same line are served from the single access (in our approach, only 4 pending loads can be served at the same cycle). This organization has been previously proposed elsewhere [11, 12, 22, 23].

Wide buses are especially attractive in the presence of vectorized loads, since multiple elements can be retrieved by a single access if the stride is small, as it is in most cases.

4. Performance evaluation

4.1. Experimental framework

For the performance evaluation we have extended the SimpleScalar v3.0a [5], which is a cycle-level simulator of a dynamically scheduled superscalar processor, to include the microarchitectural extensions described above.

To evaluate the mechanism we use 2 superscalar configurations with different issue width: 4-way and 8-way. To evaluate the memory impact of the wide bus and the dynamic vectorization mechanism we will use configurations with 1, 2 and 4 L1 data cache ports (scalar or wide). Other parameters of the microarchitecture are shown in Table 1.

Parameter	4-way	8-way
Fetch width	4 instructions (up to 1 taken branch)	8 instructions (up to 1 taken branch)
I-cache	64Kb, 2-way set associative, 64 byte lines, 1 cycle hit, 6 cycle miss time	64Kb, 2-way set associative, 64 byte lines, 1 cycle hit, 6 cycle miss time
Branch Predictor	Gshare with 64K entries	Gshare with 64K entries
Inst. window size	128 entries	256 entries
Scalar functional units (latency in brackets)	3 simple int(1); 2 int mul/div (2 for mult and 12 for div); 2 simple FP(2); 1 FP mul/div (4 for mult and 14 for div); 1 to 4 loads/stores	6 simple int(1); 3 int mul/div (2 for mult and 12 for div); 4 simple FP(2); 2 FP mul/div (4 for mult and 14 for div); 1 to 4 loads/stores
Load/Store queue	32 entries with store-load forwarding	64 entries with store-load forwarding
Issue mechanism	4-way out of order issue loads may execute when prior store addresses are known	8-way out of order issue loads may execute when prior store addresses are known
D-cache	64KB, 2-way set associative, 32 byte lines, 1 cycle hit time, write-back, 6 cycle miss time up to 16 outstanding miss	64KB, 2-way set associative, 32 byte lines, 1 cycle hit time, write-back, 6 cycle miss time up to 16 outstanding miss
L2 cache	256Kb,4-way set associative, 32 byte lines, 6 cycles hit time, 18 cycle miss time	256Kb,4-way set associative, 32 byte lines, 6 cycles hit time, 18 cycle miss time
Commit width	4 instructions	8 instructions
Vector registers	128 registers of 4 64-bit elements each	128 registers of 4 64-bit elements each
Vector functional units (latency in brackets)	Pipelined; 3 simple int(1); 2 int mul/div (2 mult, 12 div); 2 simple FP(2); 1 FP mul/div (4 mult and 14 div); 1 to 4 loads	Pipelined; 6 simple int(1); 3 int mul/div (2 mult, 12 div); 4 simple FP(2); 2 FP mul/div (4 mult and 14 div); 1 to 4 loads
TL	4-way set assoc. with 512 sets	4-way set assoc. with 512 sets
VRMT	4-way set assoc. with 64 sets	4-way set assoc. with 64 sets

***Table 1.** Processor microarchitectural parameters.*

For the experiments we use the complete SpecInt95 benchmark suite and four SpecFP95 benchmarks (swim, applu, turb3d and fpppp). Programs were compiled with the Compaq/Alpha compiler using –O5 –ifo –non_shared optimization flags. Each program was simulated for 100 million instructions after skipping the initialization part.

We have chosen vector registers with 4 elements because the average vector length for our benchmarks is relatively small: 8,84 for SpecInt and 7,37 for SpecFP applications.

For both configurations, the size of the required additional resources is the same:

- The vector register file requires 4 kilobytes (4 element per vector register * 8 bytes per element * 128 vector registers).
- The VRMT requires 4608 bytes (4 ways * 64 elements per way * 18 bytes per element).
- The TL requires 49152 bytes (4 ways * 512 elements per way * 24 bytes per element).

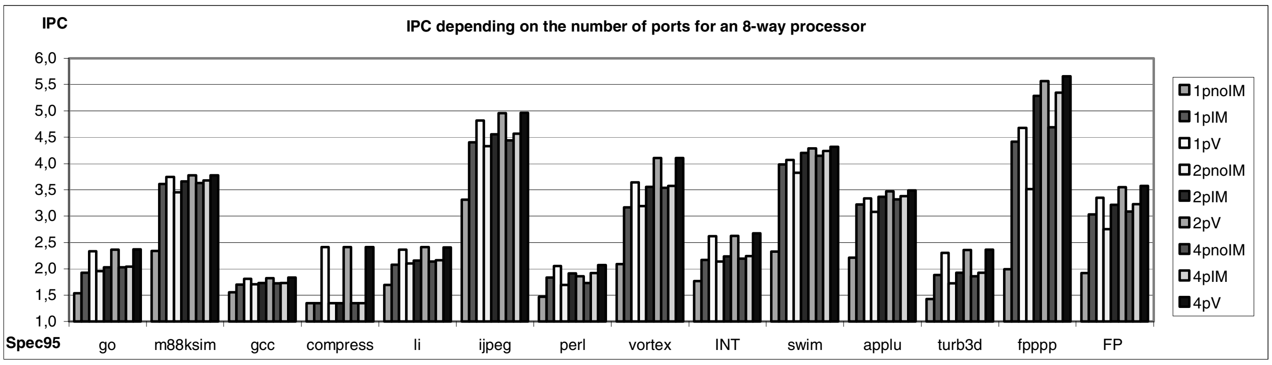
This results in a total of 56Kbytes of extra storage. Although this is not negligible this is certainly affordable in current designs.

4.3. Performance results

Figure 11 shows the performance for the 8-way and 4-way processors depending on the number of ports to L1 data cache. Each figure compares the performance of a superscalar processor (xpnoIM), a superscalar processor with a wide bus (xpIM) and a superscalar processor with a wide bus and dynamic vectorization (xpV) for a different number (x) of L1 data cache ports.

As shown in Figure 11, in most cases the configurations with wide buses increase clearly the performance of the configurations with scalar buses. The main reason is the bottlenecks due to the memory systems in configurations like an 8-way superscalar processor with 1 scalar bus. For this configuration the average IPC increased from 1,77 to 2,16 when a wide bus substitutes the scalar bus. The benefits for the configurations with 2 or 4 scalar buses are smaller since they already have a significant memory bandwidth.

Speculative dynamic vectorization reduces the pressure on the memory ports as shown in Figure 12. This Figure shows the memory port occupancy for the different processor configurations.



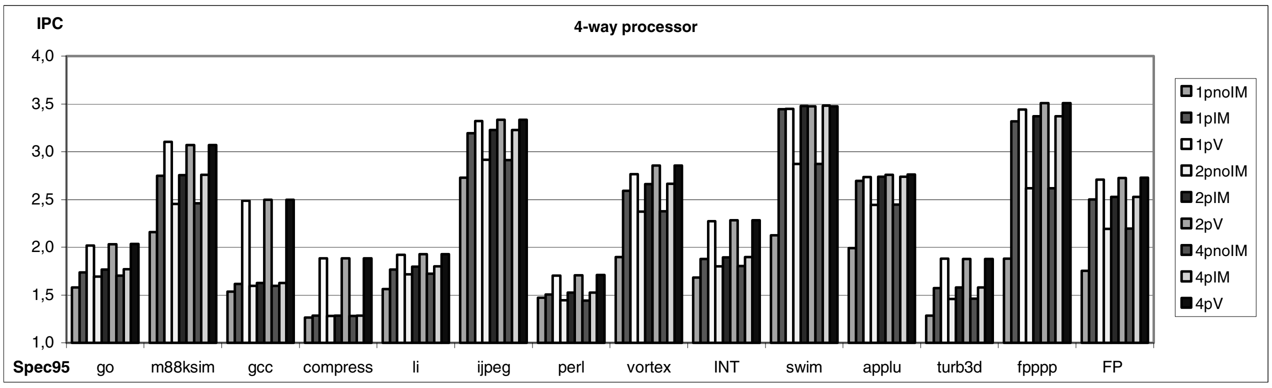
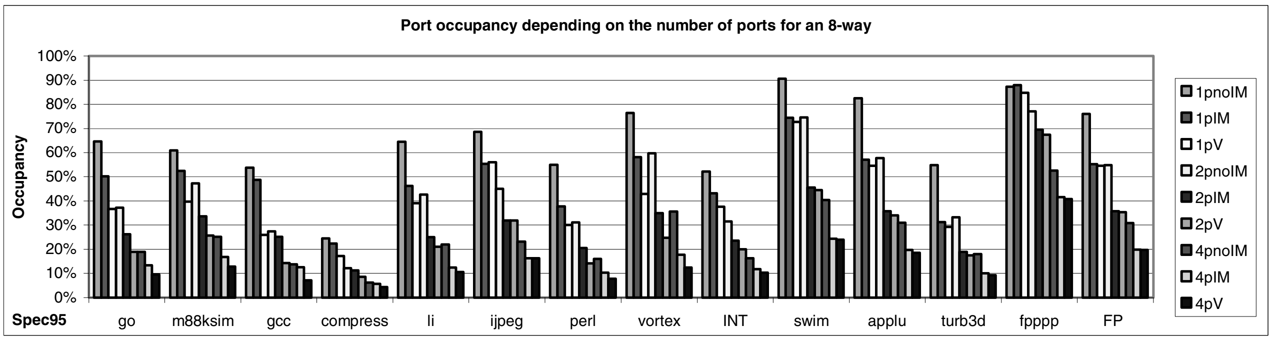


Figure 11. *IPC for the baseline configuration (xpnoIM), wide buses (xpIM), wide buses plus dynamic vectorization (xpV), for different number (x) of L1 data cache ports for an 8-way and a 4-way processor.*



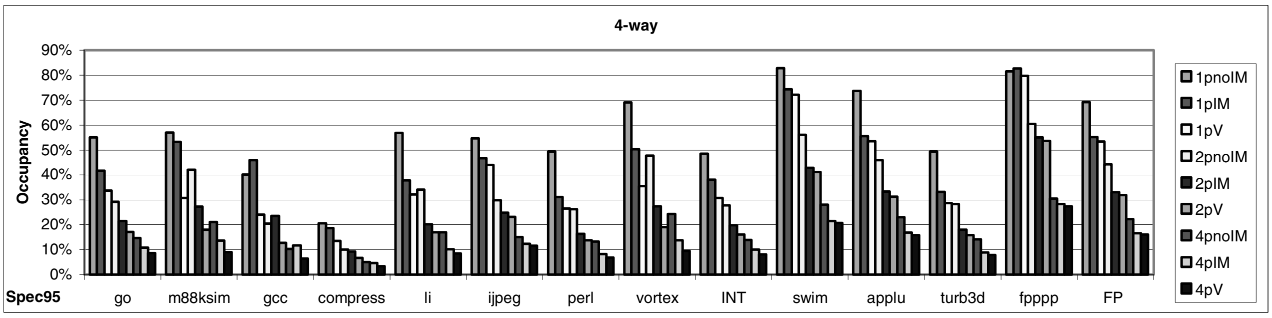


Figure 12. *Bus occupancy for the baseline configuration (xpnoIM), wide buses (xpIM), wide buses plus dynamic vectorization (xpV), for different number (x) of L1 data cache ports.*

Speculative dynamic vectorization increases the memory elements that need to be read/written due to misspeculations. However, this is shadowed by the increase in the effectiveness of the wide ports, since most vector instructions have a small stride and thus, multiple elements can be read in a single access.

Figure 13 shows the percentage of memory lines read from cache that contribute with 1, 2, 3 or 4 useful words, and the percentage of speculative (unused) accesses. It can be observed that a significant percentage of memory accesses serve multiple words and the number of useless accesses is relatively small except for compress.

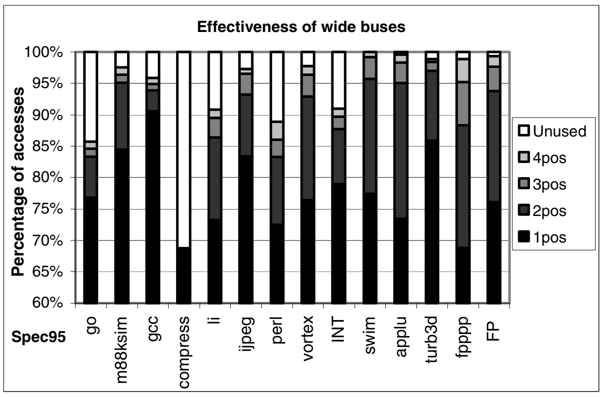


Figure 13. *Percentage of read lines that contribute with 1,2,3 or 4 useful words for a 4-way processor with 1 memory port.*

Figure 14 shows the percentage of scalar instructions that are turned into a validation operation for an 8-way superscalar processor with one wide bus. These instructions represent 28% and 23% of the total instructions for the integer and FP benchmarks respectively.

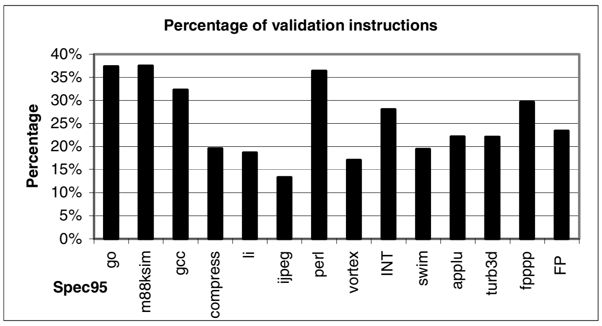


Figure 14. *Percentage of instructions that validate a position in a vector register for an 8-way superscalar processor with one wide bus and the dynamic vectorization mechanism.*

Figure 15 shows the average number of vector elements that have been computed by the vector functional units and validated (comp. used), have been computed but not validated (comp. not used) and have not been computed (not comp) for an 8-way processor with 128 vector registers.

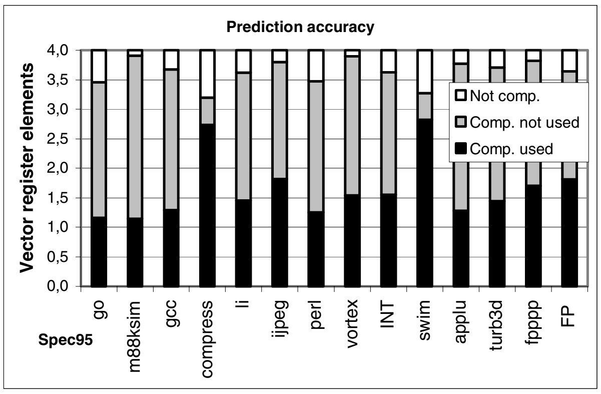


Figure 15*. Average number of vector elements that have been computed and used (comp. used), computed but not used (comp. not used) and not computed (not comp.) for an 8-way processor with 128 vector registers with 4 elements per register.*

On average only 1,75 elements are validated out of the 3,75 computed elements. This means that more than half of the speculative work is useless due to mispredictions. Although these misspeculations have a minor impact on performance, there may be an issue for power consumption. Reducing the number of misspeculations is an area that is left for future work.

As described in section 3.6, the recovery mechanism does not squash vector instructions after branch mispredictions. Due to this, the control-flow independence instructions can reuse the data computed in vector registers. As shown in Figure 10, among the first 100 instructions following a mispredicted branch (this suppose the 10,53% of total executed instructions for SpecInt95 on a 4-way superscalar processor with 1 bus and a gshare branch predictor with 64K entries), 17% of them can reuse the data stored in vector registers.

To summarize, speculative dynamic vectorization results in significant speedups that are mainly due to: a) the exploitation of SIMD parallelism, b) the ability to exploit control-flow independence, and c) the increase in the effectiveness of wide buses.

5. Related work

Vajapeyam [21] presents a dynamic vectorization mechanism based on trace processors. The mechanism executes in parallel some iterations of a loop. This mechanism tries to enlarge the instruction window capturing in vector form the body of the loops. The whole loop body is vectorized provided that all iterations of the loop follow the same control flow. The mechanism proposed in this paper is more flexible/general in the sense that it can vectorize just parts of the loop body and may allow different control flows in some parts of the loop.

The CONDEL architecture [20] proposed by Uht captures a single copy of complex loops in a static instruction window. It uses state bits per iteration to determine the control paths taken by different loop iterations and to correctly enforce dependences.

The use of wide buses has been previously considered to improve the efficiency of the memory system for different microarchitectures [12, 22, 23].

Rotenberg et al. present a mechanism to exploit control flow independence in superscalar [14] and trace [15] processors. Their approach is based on identifying control independent points dynamically, and a hardware organization of the instruction window that allows the processor to insert the instructions after a branch misprediction between instructions previously dispatched, i.e., after the mispredicted branch and before the control independent point.

Lopez et al. [11] propose and evaluate aggressive wide VLIW architectures oriented to numerical applications. The main idea is to take advantage on the existence of stride one in numerical and multimedia loops. The compiler detects load instructions to consecutive addresses and combines them into a single wide load instruction that can be efficiently executed in VLIW architectures with wide buses. The same concept is applied to groups of instructions that make computations. In some cases, these wide architectures achieve similar performance, compared to architectures where the buses and functional units are replicated, but at reduced cost.

6. Conclusions

In this paper we have proposed a speculative dynamic vectorization scheme as an extension to superscalar processors. This scheme allows the processor to prefetch data into vector registers and to speculatively manipulate these data through vector arithmetic instructions. The main benefits of this microarchitecture are a) the use of SIMD parallelism, even in irregular codes; b) the exploitation of control-flow independence; and c) the increase in the effectiveness of wide memory buses.

We have shown that these benefits result in significant speedups for a broad range of microarchitectural configurations. For instance, a 4-way superscalar processor with one wide port and speculative dynamic vectorization is 3% faster than an 8-way superscalar processor with 4 scalar ports. Speculative dynamic vectorization increases the IPC of a 4-way superscalar processor with one wide bus by 21,2% for SpecInt and 8,1% for SpecFP.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain and the European Union

(FEDER funds) under contract TIC2001-0995-C02-01 and by a research grant from Intel Corporation. We also acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for our experiments.

References

[1] J. R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Forms", *in ACM Transactions on Programming Languages and Systems, Vol. 9, no. 4,* October 1987, pp. 491-452.

[2] K. Asanovic. "Vector Microprocessors". *Phd Thesis. University of California,Berkeley*, Spring 1998.

[3] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", *in Proceedings of the 32nd Symposium on Microarchitecture*, Nov. 1999.

[4] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler Transformations for High Performance Computing", *Technical Report No. UCB/CSD-93-781, University of California-Berkeley*, 1993.

[5] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0*", Technical Report No. CS-TR-97-1342, University of Wisconsin-Madison,* Jun. 1997.

[6] R. Espasa. "Advanced Vector Architectures", *PhD Thesis. Universitat Politècnica de Catalunya, Barcelona*, February 1997.

[7] J. González, "Speculative Execution Through Value Prediction", *PhD Thesis, Universitat Politècnica de Catalunya*, January 2000.

[8] J. González and A. González. "Memory Address Prediction for Data Speculation", *in Proceedings of Europar97, Passau(Germany), August 1997.*

[9] K. Kennedy, "A Survey of Compiler Optimization Techniques," *Le Point sur la Compilation*, (M. Amirchahy and N. Neel, editors), INRIA, Le Chesnay, France, (1978), pages 115-161.

[10] Corinna G. Lee and Derek J. DeVries. "Initial Results on the Performance and Cost of Vector Microprocessors", *in Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture, Research Triangle Pk United States*. December 1 - 3, 1997.

[11] D. López, J. Llosa, M. Valero and E. Ayguadé, "Widening Resources: A Cost-Effective Technique for Aggressive ILP Architectures", *in Proceedings of the 31st International Symposium on Microarchitecture, pp 237-246,* Nov-Dec 1998.

[12] J. A. Rivers, G. S. Tyson, E. S. Davidson and T. M. Austin, "On High-Bandwidth Data Cache Design for Multi-Issue Processors", *in Proceedings of the 30th Symposium on Microarchitecture*, 1997.

[13] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors", *in 29th Fault-Tolerant Computing Symposium*, June 1999.

[14] E. Rotenberg, Q. Jacobson and J. Smith, "A Study of Control Independence in Superscalar Processors", *in Proceedings of the 5th International Symposium on High Performance Computing Architecture*, January 1999.

[15] E. Rotenberg and J. Smith, "Control Independence in Trace Processors*", in Proceedings of 32nd Symposium on Microarchitecture*, January 1999.

[16] R. M. Russell, "The Cray-1 Computer System", *in Communications of the ACM, 21(1) pp 63-72*, January 1978.

[17] J. E. Smith, G. Faanes and R. Sugumar, "Vector Instructions Set Support for Conditional Operations", *in Proceedings of the 27th Symposium on Computer Architecture*, 2000.

[18] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse*", in Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[19] J. Tubella and A. González, "Control Speculation in Multithread Processors through Dynamic Loop Detection", *in Proceedings of the 4th International Symposium on High-Performance Computer Architecture, Las Vegas (USA),* February, 1998.

[20] A. K. Uht, "Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream*", IEEE Transactions on Computers, vol. 41*, July 1992.

[21] S. Vajapeyam, J.P. Joseph and T. Mitra, "Dynamic Vectorization: A Mechanism for Exploiting Far-Flung ILP in Ordinary Programs", *in Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[22] S. W. White and S. Dhawan, "Power2", *in IBM Journal of Research and Development, v.38, n. 5, pp 493-502,* Sept 1994.

[23] K. M. Wilson and K. Olukotun, "High Bandwidth On-Chip Cache Design", *IEEE Transactions on Computers, vol. 50, no. 4*, April 2001.

[24] H. P. Zima and B. Chapman, "Supercompilers for Parallel and Vector Processors", *in ACM Press Frontier Series/Addison-Wesley*, 1990.