REDUCING THE TECHNOLOGICAL GAP BETWEEN AN ADVANCED PROCESSOR AND THE MEMORY HIERARCHY SYSTEM

***Adrian FLOREA*[*)]*, Colin EGAN*[**)]

[*)]"Lucian Blaga" University of Sibiu, Department of Computer Science, Str. E. Cioran, No. 4, Sibiu-2400, ROMANIA, E-mail: aflorea@vectra.sibiu.ro

[**)] University of Hertfordshire, Department of Computer Science, Hatfield, College Lane, AL10 9AB, UK, E-mail: C.1.Egan@herts.ac.uk

Abstract

This paper discusses dependence collapsing, instruction bypassing and 'Multiple Load' issue as advanced mechanisms to improve processor performance in superscalar architectures. We also quantify, by Trace Driven Simulations, the impact on processor performance of associating a Victim Cache/Selective Victim Cache with a first-level cache, both integrated into a parallel architecture.

1. INTRODUCTION

Exploiting and increasing instruction level parallelism (ILP) can be achieved by several approaches:

- Sophisticated superscalar processors can extract parallelism dynamically by speculative execution or by issuing instructions out-of-order.
- Instruction schedulers can achieve parallelism at compile time by reordering code, by memory reference disambiguation, and by procedure in-lining, etc.
- By collapsing dependencies between instructions to resolve Read-After-Write (RAW) hazards, which are alternatively termed true data dependencies, using ALU functional units with more than two entries.
- By technological and architectural improvements to the memory hierarchy system.

This paper focuses on the last two ideas, the removal of RAW hazards and improvement to the memory hierarchy system. In section 2, we discuss mechanisms by which RAW hazards can be reduced. In section 3, we quantify the association of a Victim Cache/Selective Victim Cache with a first-level cache. In the experimental aspects of our study, we use the Hatfield Superscalar Architecture (HSA) that has been developed, as a convenient architectural vehicle for research into ILP [3], at the University of Hertfordshire, UK.

2. REDUCING RAW HAZARDS

RAW hazards occur when a destination operand of one instruction is required as a source operand for a second instruction. RAW hazards can be removed by combining the data dependencies between the two instructions, thus collapsing the dependence between the instructions [7, 11, 12, 15]. **Dependence collapsing** was originally proposed as a hardware technique, termed Interlock Collapsing by Vassiliadis at IBM [15]. Subsequently, a software version of Interlock Collapsing, which has been termed Instruction Combining, was developed at the University of Hertfordshire [12], using the Hatfield Superscalar Scheduler (HSS). During the process of dependence collapsing a pair of dependent instructions, which do not need to be adjacent to each other, are combined into a single complex instruction.

In a previous study [7] we implemented dynamic dependence collapsing with the use of a prefetch buffer. In the case of a single-port Data Cache, processor performance was improved by 11.19%, when a two-port Data Cache was used there was an 8.51% improvement, reaching a maximum value of 26.34% on some benchmarks [7]. Additionally, we showed that increasing the number of collapsible instructions from 2 to 3 improved processor performance by approximately 22%. Further, increasing the number of collapsible instructions to 4 resulted in a 20% increase in the number of combined instructions. However, increasing the number of collapsible instructions to 6, only achieved marginal processor performance improvement.

A small **Data Write Buffer** (DWB) [16] can also be used to resolve RAW hazards. The main idea of a DWB buffer is to prevent processor stalls from occurring when there are insufficient cache write ports available to service the requests for data writes. The DWB works in parallel with the processor, releasing the processor from writing data to the Data Cache. An entry in the DWB contains the virtual address of an instruction and the data that must be written to the Data Cache. Consider the following case, assuming that the DWB has sufficient ports for supporting the worst case situation, such as a multitude of Store instructions. Also assume, that the Store instructions are independent, concurrent and stored in an *instruction window*. Since, the DWB is a small fast cache adding sufficient virtual write ports to satisfy peak demand is simple and cheap [16]. The Data Cache therefore only requires sufficient write ports to satisfy the average number of writes per clock cycle. Consequently, the Data Cache requires one or, at maximum, two read ports and a single write port. In our previous studies [7, 12], we showed that increasing the latency of a DWB from 2 to 3 clock cycles reduced processor performance by 11.63%. We concluded that writing from the DWB to the Data Cache must be conducted quickly to minimise impact of latency. We also showed that increasing the number of read ports to 2 improved processor performance by 5%.

Bypassing, or forwarding, can be used to remove Load and Store hazards. In pipelined processors, results obtained after the execution stage are returned to the register file. However, in bypassing, a result may be forwarded directly to the execution stage of the pipeline if the result is required as a source operand by an immediately following operation. In our earlier studies [7, 12], when bypassing was included with a DWB we showed that processor performance improved by 17.87%.

Reducing the cache miss penalty improves processor performance [14]. Assuming that target addresses of Load and Store instructions in the Instruction Buffer are known, then multiple Load instructions can be issued. The '**Multiple Loads**' are read from the same Data Cache block, providing there are no intermediate Stores. Issuing Multiple Loads can be applied out-of-order, where reservation stations are provided for Load and Store instructions, or in a Trace Processor [13]. The main advantage of Multiple Loads is an ability to execute

multiple loads in each cycle without fully replicating the cache read port. However, the size of the Data Cache and the size of the Instruction Buffer directly influence the processing rate when Multiple Loads is used. For example, with a small Data Cache that has few blocks, there is a high possibility that the same block is accessed repeatedly. In the case of a large Instruction Buffer, there is a high possibility that more Loads accessing the same Data Cache block are retained. In our previous studies [7, 12], we showed that Multiple Loads only improves processor performance by about 3%, and that any improvement is dependent on the execution pattern of the instructions.

Throughout the 1990s, processor cycle time has been decreasing at a rate faster than the time to access the memory system. Additionally, the architectural design of processors has improved with the development of pipelined and multiple instruction issue architectures. These factors have influenced the increasing technological gap between processor speed and the speed of the underlying memory hierarchy, and have dramatically reduced the average number of clock cycles per instruction (CPI) [14].

The memory access time is a critical factor in system design. A slow memory access time can be a cause of a major bottleneck. Increasing memory access time and therefore reducing the bottleneck can be achieved by organising the memory system in a hierarchy. This hierarchy involves implementing a fast SRAM cache on the CPU chip, with a slower main DRAM behind it. Access time to current state-of-the-art SRAM caches is between 1 and 7 ns. In this study we have chosen to use Harvard direct-mapped split Instruction and Data Caches in preference to set-associative caches. We made this design decision for the following five reasons. First, direct-mapped caches are simple to implement. Second, a direct-mapped cache has a lower cost than a set-associative cache. Third, for large cache sizes a direct-mapped cache has a quicker effective average access time than a set associative cache. Fourth, it is possible to integrate a direct-mapped cache directly onto the CPU [1]. Finally, small direct-mapped caches have a one clock cycle access time. However, one of the disadvantages of direct-mapped caches is that they make parallel address translation more difficult.

Researchers into multiple instruction issue (MII) usually assume that a processor has a perfect memory hierarchy and that the caches have a perfect hit rate. Therefore there are no cache misses. In our study, we do not make these assumptions; instead we assume that we have a perfect processor model that has an imperfect memory system. With this system design, we can now investigate the limits that caches have on the instruction issue. In a previous study using trace driven simulations [8], we showed that the average issue rate of an imperfect memory system and an imperfect cache hit rate was approximately 60% of the average issue rate of a model with a perfect memory system and with a perfect cache hit rate. We concluded that there is a reduction in processor performance due to miss reading or writing processes in the cache.

3. ASSOCIATING A VICTIM CACHE/SELECTIVE VICTIM CACHE

Jouppi [2] proposed the use of a Victim Cache to reduce the miss rate of direct-mapped caches. A Victim Cache is a small fully associative cache that is positioned between the first-level cache and the main memory. The Victim Cache does not affect the first-level cache hit rate or its miss penalty. The Victim Cache stores blocks that are displaced from the first-level cache. If a block is referenced again before being replaced from the Victim Cache,

it can be fetched directly from the Victim Cache. This results in a far lower penalty than a block that is brought in from main memory [2].

To minimise the number of interchanges between the first-level cache and the Victim Cache, Stiliadis [4] proposed the use of a Selective Victim Cache, see figure 1. Incoming blocks from main memory are either placed in the first-level cache or in the Selective Victim Cache. The placing of a block in either the first-level cache or in the Selective Victim Cache is by a prediction mechanism that depends on the previous use history of the block. The prediction algorithm is also used in the case of a first-level cache miss to determine if an interchange of conflicting blocks is appropriate. The main idea of the prediction algorithm is to place blocks that are likely to be referenced again in the first-level cache, and those that are not likely to be referenced again in the Selective Victim Cache.

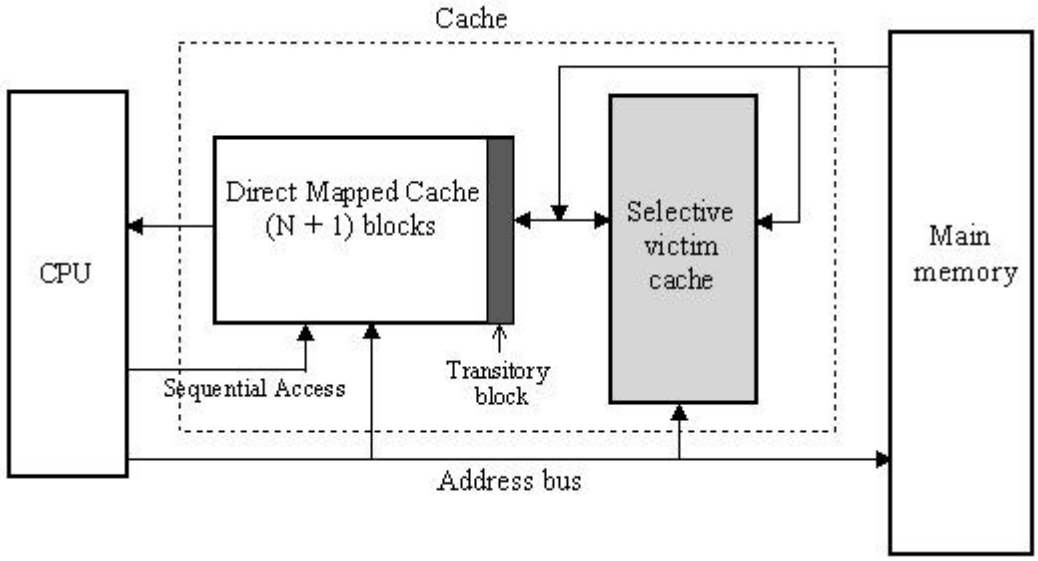

Figure 1 Memory hierarchy for the Selective Victim Cache.

Every cache block has two state bits, a *hit-bit* and a *sticky-bit*. The prediction algorithm uses these state bits to determine where a block is to be placed. Previous history information is contained in the hit-bit, which is only set if a cache block has previously been in the first-level cache and that block has been accessed. In the case of a block being present in the first-level cache but not accessed the hit-bit is not set. When a block is replaced from the first-level cache, the state of the corresponding hit-bit is also updated in main memory. When a block is fetched into the first-level cache, its sticky-bit is set. On a reference to a conflicting block, if the block is not replaced from the first-level cache then the sticky-bit is reset. On subsequent first-level cache accesses, if a conflict reoccurs with the same block then it is replaced [4].

A Selective Victim Cache is implemented by increasing the size of the first-level cache by one block, which is termed the ‘transitory block’. The prediction algorithm uses the transitory block for sequential references of the same block. For example, when a hit occurs in the Selective Victim Cache and the prediction algorithm selects not to perform an interchange, then the corresponding block is copied from the Selective Victim Cache into the transitory block. In this case, the transitory block acts as a buffer so that sequential references to the block can be served. Sequential references to a block are repeated accesses to the block without any intervening references to other blocks. Sequential references are recognised by

using a Sequential Access signal, which is activated by the CPU whenever the current memory reference accesses the same block as the previous memory reference.

In figure 2, case 1 and case 2 are two different examples of the Selective Victim Cache prediction algorithm. The bold text, of lines *1.1* and *2.1*, highlights the modifications that we suggest could be used to improve the prediction algorithm. With our modifications there is no reason to set the hit-bit of a new block until a **true-hit** has occurred. A true-hit occurs after there has been a first-level cache hit. Another modification would be to consider hit and sticky vectors instead of these simple hit/sticky-bits.

```
Case 1: Access to block B, hit in selective victim cache
          A is the conflicting block in main cache

        if sticky[A] == 0 then
                interchange A and B
        1.1     sticky[B] ← 1;  hit[B]  ← 1 (hit[B] ¬  0)
                copy B to transitory block
        else
                if hit[B]= = 0 then
        1.2             sticky[A] ← 0;
                        copy B to transitory block
                else
        1.3             interchange A and B
                        sticky[B] ← 1; hit[B] ← 0;

Case 2: Access to block B, miss in both main and selective victim caches
          A is the conflicting block in main cache

        if sticky[A] == 0 then
                move A to victim cache
                transfer B to main cache
        2.1     sticky[B] ← 1;  hit[B]  ← 1 (hit[B] ¬  0), etc.
```

Figure 2 The Selective Victim Cache prediction algorithm.

4. THE SIMULATION METHOD

We use trace driven simulations to evaluate the impact of our cache organisation on processor performance. We quantify issue rate, first level cache hit/miss rate and Selective Victim Cache usage on a range of cache organisations. We also investigate the impact that a Victim Cache and a Selective Victim Cache has on processor performance. We use the Stanford Integer benchmarks that are a suite of eight 'C' programs representing general-purpose programs. These benchmarks are computationally intensive with high dynamic instruction counts, and are heavily recursive. The dynamic instruction distribution of the benchmarks is fairly typical: 53% are Arithmetical, Logical, Shifts or Relational, 18% are Loads, 12% are Stores and 17% are Branches [7, 8]. The benchmarks were compiled by the

HSA Gnu C compiler, which targets the HSA instruction set. A dedicated HSA instruction level simulator [3] executed the resultant HSA object code, which in turn was used to generate the corresponding trace files that we used in our simulations. We have developed a trace driven simulator that uses these trace files. Our trace driven simulator permits different run time configurations to test various model parameters:

- **FR**: is the fetch rate. This is a variable number of instructions that are fetched from Instruction Cache, up to a maximum of 16 instructions per cycle (IPC).
- **IBS**: is the Instruction Buffer size that can contain up to 64 instructions.
- **Size of Main Cache Memories**: provides up to 16K locations. In our study we use a direct-mapped split Instruction and Data Cache structure. The size of the caches is measured by location. The Instruction Cache stores instructions, and the Data Cache stores memory addresses.
- **Victim Cache**: can contain up to 64 locations.
- **Selective Victim Cache**: can contain up to 64 locations.
- **IRmax**: is the maximum number of instructions that can be issued in parallel. The number of instructions that can be dispatched from the Instruction Buffer is up to 8 IPC.
- **MP**: is the miss penalty. The number of clock cycles required to load a block from main memory can be varied up to 20-clock cycles.
- **VC_delay**: is the penalty for reading from Victim Cache if there is no interchange. This can be varied up to 3 cycles and is dependent on the size of the Victim Cache.
- **Interchange Ticks**: this is the penalty for interchanging a block. It is fixed at 3 cycles.

Although, the first-level cache is direct-mapped, the Victim Cache/Selective Victim Cache is fully associative. We have implemented two replacement algorithms: the Least Recently Used (LRU) and Random Replacement algorithm. We have also implemented a single port Data Cache that can be accessed by only one Load/Store instruction.

In our simulations, there is latency of 3 cycles for every interchange operation between the first-level cache and the Victim Cache/Selective Victim Cache. There is also an additional access penalty of 1 cycle for every reference serviced by the Victim Cache. This additional penalty is realistic when a Memory Management Unit (MMU), to translate virtual and physical addresses, is considered. For example, a hit serviced by the Victim Cache requires 2 cycles, one for the MMU address translation, virtual address to physical address, and one to access the Victim Cache. Furthermore, copying a block from the Selective Victim Cache to the transitory block involves an additional penalty.

The main metric we use in our simulations is the *average instruction issue rate*. This encapsulates all simulated processes and generates a realist performance indicator. We also quantify cache hit/miss rates and the number of interchanges between first-level cache and Victim Cache. We therefore quantify a more comprehensive range of statistics than Stiliadis [4], who only quantified the miss rate. We have used these statistics to compute average memory access times.

5. OBTAINED RESULTS

We simulated a Harvard Superscalar Architecture that has a split Instruction Cache and Data Cache. The direct-mapped Instruction Cache was set to have a latency of 2 cycles, and the Data Cache to 3 cycles. We set the miss penalty to 15 cycles. We also used the

following parameters: a fetch rate (FR) of 8 instructions, an Instruction Buffer (IBS) to hold a maximum of 32 instructions, and an instruction issue rate to 4 IPC. Our processor model also had one Load/Store execution unit. The fully associative Victim Cache had 32 locations or the alternative Selective Victim Cache had 8 locations.

Figure 3 shows the harmonic mean *issue rates*, of the Stanford benchmarks, as a function of Data Cache size. The issue rate speedup, with and without the 8 location Selective Victim Cache varies from 2.15 for a 256 entry Data Cache to 1.83 for a 4096 entry Data Cache. Similarly, the first level Instruction Cache sustains an issue rate speedup, with and without 8 locations in the Selective Victim Caches, varies from 2.45 for a 32 location Instruction Cache to 1.64 for a 128 entry Instruction Cache.

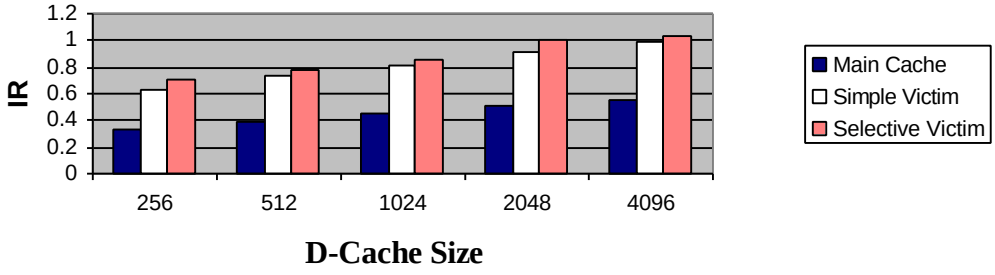

Figure 3 Issue Rate as a function of the Data Cache capacity.

Figure 4 shows the *hit rate* of the Data Cache. Also, our results show that a Selective Victim Cache improves the hit rate of Instruction-Caches with up to 64 locations. An Instruction Cache with 64 locations without a Victim Cache has a hit rate of 62%; when a Selective Victim Cache is implemented the hit rate improves to 94%.

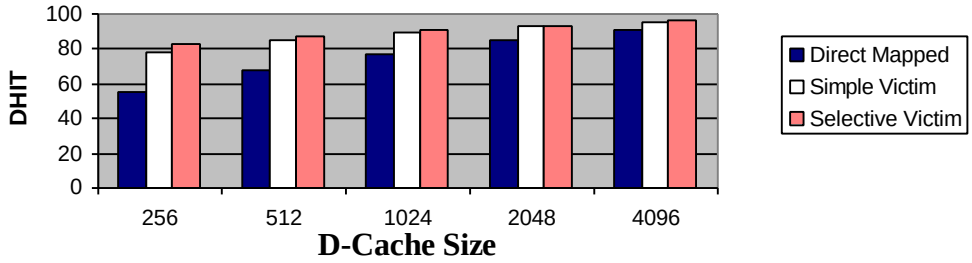

Figure 4 Data Cache hit rate with increasing capacity.

Figure 5 compares the *number of interchanges* between the Data Cache and the Victim Cache/Selective Victim Cache. The plot shows that the Selective Victim Cache has fewer interchanges than the Victim Cache for Data Caches with less than 4096 locations. For example, with a 256 location Data Cache there are 7,426 interchanges with the Victim Cache, which is reduced when the Victim Cache is replaced with a Selective Victim Cache to 4,227 interchanges. When the number of locations in the Data Cache is increased to 2,048, there are 987 interchanges with the Victim Cache and only 680 interchanges with the Selective Victim Cache. The reduction in the number of interchanges reflects the increase in size of the Data Cache and the increased hit rate. When the number of locations in the Data Cache is increased to 4,096, the number of interchanges between Data Cache and the Victim Cache/Selective

Victim Cache is approximately the same. Again this is reflected by the hit rate of the Data Cache.

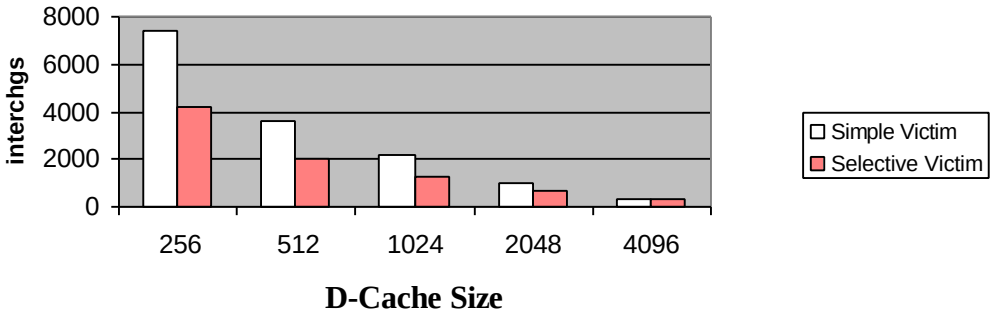

Figure 5 The number of interchanges between the Victim Cache/Selective Victim Cache and increasing Data Cache size.

We also quantified the *utilisation rate* of the caches. The results showed that the utilisation rate of a Data Cache with 256 locations is almost 100%, and the utilisation rate of its associated Victim Cache/Selective Victim Cache is 88.67%. These results suggest that an optimal Data Cache has 2,048 locations. In our study an optimal 2K Data Cache has an utilisation rate of 63.82%, and the utilisation rate of its Victim Cache/Selective Victim Cache is 49.2%. A similar quantitative analysis shows us that the optimal **I-Cache** size is about **128** entries (associated with a 8 entries victim cache), at an utilisation level of over 97% for main cache and about 60% for the attached (selective) victim cache. For a 32 I-Cache size we obtained (arithmetic means) 12647 vs. 5701 interchanges and for a 128 I-Cache size 3927 vs. 1200 interchanges. During all the following analysis we will use the optimal values that we have found.

For the remaining experiments we have used the optimal Data and Instruction Caches to quantify the optimal Victim and Selective Victim Cache sizes. Figure 6 shows the issue rates as a function of the Victim Cache and Selective Victim Cache sizes. The plot shows that the issue rate fluctuates at around 0.8 when a Victim Cache is associated with the Data Cache irrespective of the Victim Cache size. Similarly, the issue rate fluctuates around 0.85 when a Selective Victim Cache is associated with the Data Cache irrespective of the size of the Selective Victim Cache.

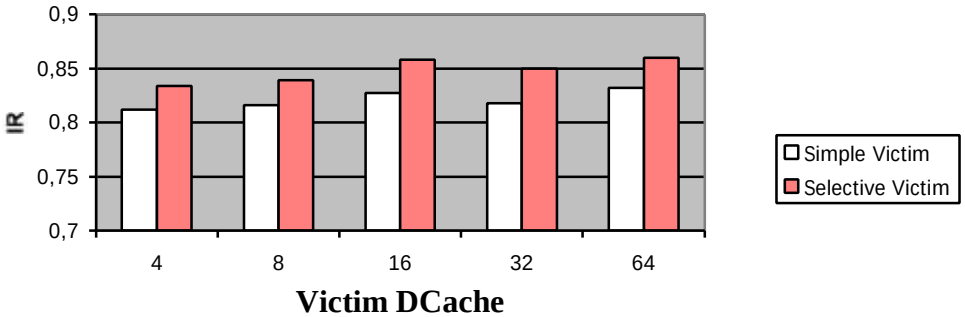

Figure 6 Processor performance for different Data (Selective) Victim Cache's capacities.

Figure 7 shows that the number of interchanges between the optimal Data Cache and the Victim Cache increases as a function of Victim Cache size. For example, when there are only 4 locations in the Victim Cache the number of interchanges with the Data Cache is about

1,000. When the number of locations in the Victim Cache is increased to 64, the number of interchanges is about 2,800. A similar pattern emerges when the Selective Victim Cache replaces the Victim Cache but, as with figure 5, there are a reduced number of interchanges compared to the Victim Cache.

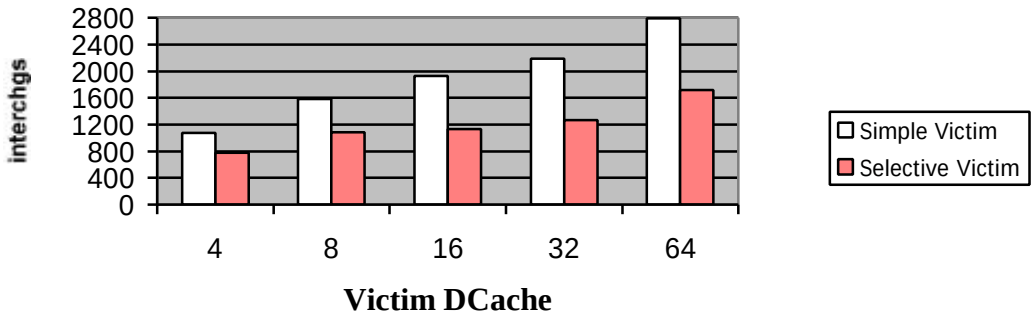

Figure 7 Number of interchanges between D-Cache and (S) VC

The SVC's main "force" consists in a serious decrement of block interchanges between main direct mapped cache and SVC itself. So, for the optimum values that we found, a 32-entry size of data victim cache together with a 2k entries direct mapped D-Cache, the interchange's number is reduced with 72 percent, that's significant. Also, for the same size of (S) VC, the utilisation rate is about 81%.

Because it's impractical to store the hit bit associated with each block in main memory like in a perfect implementation [4], we considered an alternate approach that maintain the hit bits within the processor itself, alongside the L1 cache. So, we are using a hashing hit-array scheme for storing hit bits that involve write interferences. The results obtained using ideal implementation vs. a hit array scheme are presented in the next table, pointing out that the performances are quite similar in both cases, according to [4]. Therefore interferences have little impact that permits an elegant feasible implementation of the SVC concept.

		ICache Usage	IVictim Usage	ICache interchg	IHIT %	DCache Usage	DVictim Usage	DCache interchg	DHIT %	IR
I	FBubble	100%	87.5%	9	99.97	79.69%	37.50%	2	100	1.340
D	FMatrix	100%	100%	10	99.96	100%	100%	1887	82.80	0.695
E	Fperm	100%	75%	5681	99.98	21.88%	9.38%	1	100	1.026
A	Fpuzzle	100%	100%	6006	98.46	100%	100%	1914	71.83	0.733
L	Fqueens	100%	100%	2137	99.67	50.78%	100%	2748	99.82	1.044
	Fsort	100%	100%	340	99.86	100%	100%	387	95.60	1.179
	Ftower	100%	100%	9223	70.54	32.81%	100%	658	97.95	0.576
	FTree	100%	100%	18	99.93	100%	100%	2528	83.77	0.746
	MEAN	**100%**	**95.31%**	**2928**	**94.79**	**73.14%**	**80.86%**	**1265**	**90.26**	**0.85**
H	Fbubble	100%	87.5%	9	99.97	79.69%	37.50%	2	100	1.340
A	FMatrix	100%	100%	10	99.96	100%	100%	779	82.82	0.699
S	FPerm	100%	75%	5681	99.98	21.88%	9.38%	1	100	1.026
H	Fpuzzle	100%	100%	6107	98.44	100%	100%	3230	71.13	0.723
I	Fqueens	100%	100%	2135	99.67	50.78%	100%	2800	99.82	1.051
N	FSort	100%	100%	339	99.86	100%	100%	366	95.62	1.182
G	Ftower	100%	100%	8712	70.54	32.81%	100%	1362	98.33	0.587
	FTree	100%	100%	17	99.93	100%	100%	2455	83.42	0.742
	MEAN	**100%**	**95.31%**	**2876**	**94.78**	**73.14%**	**80.86%**	**1374**	**90.12**	**0.85**

Table 1. Comparative Study about **ideal** implementation **vs.** a **hashing** hit array scheme

Finally we generalised the original algorithm proposed by Stiliadis and Varma and presented shortly by us in Section 3, by considering binary vectors instead of a simple bit, for "hit" and "sticky" states, for a better tracking of block history. We believe that a hit vector will indicate more gradually the utilisation rate to the block occurred while it was in main cache the last time. When a new block is brought in main cache, its corresponding hit vector will be initialised to zero. It will be incremented during each hit involved by that block its current value meaning the utilisation rate. Analogously, a sticky vector is associated with each block in the main cache (level one). On a reference to a conflicting block, if the prediction algorithm decides not to replace the corresponding block in the main cache, its sticky vector will be incremented by one, meaning a new access conflict related to that block (the block became more conflicting). Therefore, our algorithm will replace decisions like "if sticky [block]=0" or "if hit [block]=0" from the original algorithm presented by us in Section 3, to more gradually decisions like "if sticky [block]>half" or "if hit [block]<half", where "half" means half of the maximum hit/ sticky vectors values.

		IHIT (%)	DHIT (%)	IR
1 BIT	Fbubble	99.97	100	1.3408
	Fmatrix	99.96	82.80	0.7176
	Fperm	99.98	100	1.0353
	Fpuzzle	99.16	71.83	0.7708
	Fqueens	99.95	99.82	1.0878
	Fsort	99.88	95.60	1.2029
	Ftower	92.81	97.95	0.8339
	Ftree	99.94	84.77	0.7883
	MEAN	**98.83**	**90.41**	**0.927**
2 BITS	Fbubble	99.97	100	1.3409
	Fmatrix	99.96	82.77	0.7184
	Fperm	99.98	100	1.0353
	Fpuzzle	99.21	71.65	0.7686
	Fqueens	99.95	99.26	1.0805
	Fsort	99.88	95.71	1.2073
	Ftower	94.57	98.10	0.8823
	Ftree	99.94	83.60	0.7854
	MEAN	**99.15**	**90.17**	**0.934**
4 BITS	FBubble	99.97	100	1.0349
	fMatrix	99.96	82.36	0.7116
	fPerm	99.98	100	1.0353
	fPuzzle	99.21	70.50	0.7535
	fQueens	99.95	99.63	1.0913
	fSort	99.87	95.11	1.1875
	fTower	94.57	98.81	0.9087
	fTree	99.94	80.67	0.7437
	MEAN	**99.14**	**90.88**	**0.925**
8 BITS	fBubble	99.97	100	1.3409
	fMatrix	99.96	82.77	0.7184
	fPerm	99.98	100	1.0353
	fPuzzle	99.21	70.99	0.7592
	fQueens	99.95	99.37	1.0846
	fSort	99.88	95.72	1.2081
	fTower	94.57	99.77	0.9298
	fTree	99.94	83.87	0.7896
	MEAN	**99.15**	**90.26**	**0.939**

Table 2. Tracking the block history using **binary vectors**

Some obtained results for different vector lengths are presented in the table 2, for I-Cache =128 with SVC=4 respectively D-Cache=1024 with SVC=32. As it can be observed, the conclusion is that 2 bits hit and sticky vectors are optimal, involving a global speedup about 1% over the 1 bit states. Taking into account some hit-array schemes based on hashing, we consider feasible to implement these 2 bit arrays.

4. CONCLUSIONS

In this paper we investigated multiple-instruction-issue in a high-performance superscalar architecture, illustrating both the improvements and the limits of some well-known techniques (like dependence collapsing and instruction bypassing). Through trace driven simulation we quantify the impact of the victim cache and selective victim-caching scheme on the performance of an in-order superscalar processor, illustrating the optimum values for some processing parameters. We have also presented some modifications in the prediction algorithm that in our opinion can improve the effectiveness of the selective victim cache architecture.

As a reply to victim cache concept, proposed by Jouppi (1990), for improving the hit ratio of direct-mapped caches without affecting the access time, Stiliadis (1994) introduced Selective Victim Cache method. The method selectively places data in the main cache or the victim cache based on a prediction algorithm that determine how likely they are to be accessed in the future. Using Selective Victim Cache, the simulation results show significant improvements in terms of both issue rate and hit rate. Also, the performance improvements are highly determinate by the prediction algorithm impact over the number of interchanges between direct-mapped cache and victim cache. The algorithm may also contribute to a better placements of block in the cache, reducing the number of accesses in victim cache and generating higher hit rates in the direct-mapped cache.

Stiliadis concludes [4] that, for data caches, the effectiveness of the prediction algorithm employed in selective victim caching was reduced considerably due to the more random patterns of data references, making it much less attractive rather than implementing the selective victim cache concept for an instruction cache. However, he considers that a fully associative victim cache may still be attractive when used in a data cache. Starting by these conclusions we implemented a selective victim cache fully associative. Although the results are very encouraging – significant improvements of imposed performance metrics – they are perfectly concordance with Stiliadis conclusions (the significant improvements appear using small caches, static allocation of data and regular data structures). We think the explanation consists of following two aspects:

- The optimum data cache size, except *puzzle* (about 24K data space), for Stanford's benchmark is 2K location (=8K bytes) [8] – pretty small.
- All Stanford programs employ static allocation of data and regular data structures (arrays) the prediction algorithm being able to resolve a large number of conflicts in these cases.

Probably, in the case of programs with dynamic memory allocation and extensive use of pointers (only the *tree* benchmark combines static with dynamic allocation of data), the conflicts are much harder to resolve by the prediction algorithm. Anyway, we think the problem is still open, the solution consisting in finding an independent algorithm by random patterns of data references.

REFERENCES

[1] Hill M. – *A Case for Direct-Mapped Caches*, IEEE Computer, December 1988.

[2] Jouppi N. – *Improving Direct-Mapped Cache Performance by the addition of a Small Fully Associative Cache and Prefetch Buffers*, Proc. 17th International Symposium On Computer Architecture, 1990.

[3] Steven G. – *A Superscalar Architecture to Exploit Instruction Level Parallelism*, Microprocessors and Microsystems", No 7, 1997.

[4] Stiliadis D., **Varma A.** – *Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches*, TR UCSC-CRL-93-41, University of California, 1994 (republished in a shorter version in IEEE Trans. on Computers, May 1997).

[5] Steven G., **Vintan L.** – *Modelling Superscalar Pipelines With Finite State Machines*, Proceedings of the Euromicro '96 Conference, IEEE Computer Society Press, California, 1996.

[6] Vintan L., **Steven G.** – *Memory Hierarchy Limitations in Multiple Instruction-Issue Processor Design*, Proc. Euromicro '97, IEEE Computer Society Press, California, 1997.

[7] Florea A., **Vintan L.** – *Advanced Techniques for Improving Processor Performance in a Superscalar Architecture* - Proceedings of The 12th International Conference on Control Systems and Computer Science (CSCS 12), vol.2, Bucharest, Romania.

[8] Florea A. – *Optimizarea proceselor de scriere într-o arhitecture RISC superscalara de tip Harvard*, Master Thesis, Sibiu, 1998 (coordinator L. Vintan), in Romanian.

[9] Florea A. – *Simulator pentru o arhitecture superscalar parametrizabile de tip Princeton*, BEng Thesis, Sibiu, 1997 (coordinator L. Vintan), in Romanian.

[10] Florea A., **Vintan L.** – *Simulating Some Advanced Processing Techniques into a Superscalar Architecture* – Proceedings of The 12th International Conference Beyond 2000: Engineering Research Strategies, Sibiu, 1999.

[11] Sazeides Y., Vassiliades S. and Smith J. E. – The Performance Potential of Data Dependence Speculation & Collapsing, IEEE Transactions Computers, Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Paris 1996.

[12] Steven F. – An Introduction to the Hatfield Superscalar Scheduler, Technical Report Number 316, Computer Science, University of Hertfordshire, June 1998.

[13] Vintan L. – *Architectura procesoarelor cu paralelism la nivelul instructiunilor*, Editura Academiei Române, Bucuresti, 2000, in Romanian.

[14] Hennessy J., Patterson D. – *Computer Architecture*: *A Quantitative Approach*, Morgan Kaufmann Publishers, Second Edition, 1996.

[15] Vassiliadis S., Phillips J., Blaner B. – *Interlock Collapsing ALUs*, IEEE Transactions on Computers, Vol. 42, No. 7, 1993, pp. 825 – 839.

[16] Tate D., Steven G. - *Adding a Cache Simulator to the Hatfield Superscalar Project*, University of Hertfordshire, Technical Report, 1998.

[17] Vintan L., Armat C., Steven G.- *The Impact of Cache Organisation on the Instruction Issue Rate of a Superscalar Processor*, Proceedings of Euromicro 7th Workshop on Parallel and Distributed Systems pg. 58-65, ISBN 0-7695-0059-5, Funchal, Portugal, 3-rd -5th February, 1999