A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks

Henk Neefs, Hans Vandierendonck and Koen De Bosschere
Dept. of Electronics and Information Systems, University of Gent
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{neefs,hvdieren,kdb}@elis.rug.ac.be

Abstract

One of the problems in future processors will be the resource conflicts caused by several load/store units competing to access the same cache bank. The traditional approach for handling this case is by introducing buffers combined with a cross-bar. This approach suffers from (i) the non-deterministic latency of a load/store and (ii) the extra latency caused by the cross-bar and the buffer management. A deterministic latency is of the utmost importance for the forwarding mechanism of out-of-order processors because it enables back-to-back operation of instructions. We propose a technique by which we eliminate the buffers and cross-bars from the critical path of the load/store execution. This results in both, a low and a deterministic latency. Our solution consists of predicting which bank is to be accessed. Only in the case of a wrong prediction a penalty results.

1. Introduction

The instruction level parallelism (ILP) that out-of-order processors extract keeps on growing. Architectural techniques like predication, memory dependence speculation and data value speculation all increase the attainable ILP. As a consequence, more than the contemporary two load/store units will be required in the near future. It can be anticipated that soon, 4 or more load/store units will be needed. This fact, together with the steadily increasing clock speeds, necessitate very high bandwidth caches. High bandwidth caches can be realized through the use of pipelined caches [1], multiple cache banks or a combination of both. In this paper we focus on the multiple banks approach.

With multiple banks the question arises how the interconnection network (Figure 1) between the load/store units and the caches banks should look like. Ideally, the interconnection network should allow a high bandwidth, have a low latency and this latency should be deterministic. The latter is required to enable efficient back-to-back execution of dependent instructions in an out-of-order processor with forwarding.

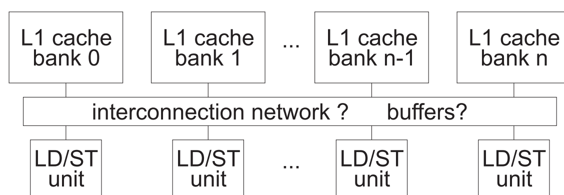


Figure 1. The scrutinized problem.

When the load/store latency is non-deterministic, i.e. sometimes takes on a higher value, dependent instructions could have been selected. These instructions cannot execute yet because the results will arrive later than anticipated. So part of the processor will stall. Alternatively, the selected instructions could be nullified and reselected at some later time. The latter technique is used in the Alpha 21264 [1] when an L1 cache miss is detected (and a miss was not predicted); the instructions selected for execution in the previous two cycles are nullified and tagged reselectable in the instruction window.

In the next section we will describe some traditional solutions to the sketched problem and introduce a new and better solution: bank prediction. Since our approach requires prediction, several known prediction algorithms will be investigated in section 4. Since the ultimate measure of a solution is the performance of the processor, we present processor performance figures in section 5.

2. Some solutions

To state the problem again, we investigate what the L1-cache and more in particular the L1-cache to load/store units interconnection network should look like to provide high bandwidth simultaneous accesses by several load/store units.

2.1. The traditional approach

Multiple cache banks can pose as a high bandwidth cache if the interconnection network between load/store-units and banks supports this. A traditional solution to this interconnection problem is presented in Figure 2. Through a cross-bar, every load/store unit can access every cache bank. Furthermore several load/store units can attempt to access the

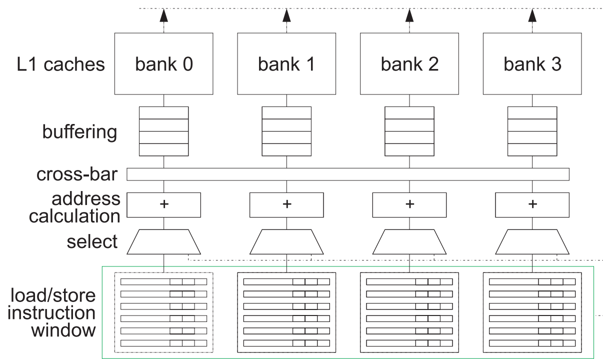


Figure 2. Traditional approach.

same bank in one single cycle. To cover this case, buffering is introduced. Remark that on the one hand, dividing the cache in several banks results in lower latency cache-bank accesses, but on the other hand extra latency is introduced by the cross-bar (especially by the buffertree which has to drive numerous multiplexers in the cross-bar) and the buffer management hardware. On top of this, the cross-bar and buffer-hardware latency increase with an increasing number of load/store units. Furthermore, because of the buffering, the latency of the load/store units is no longer deterministic, which again has a negative impact on performance. Therefore another solution with lower deterministic latency and better scalability should be found. This would be possible when the cross-bar and the buffering could be eliminated in some way.

2.2. The apparent solution

In Figure 3 an apparently simple solution is presented. Based on the memory address, one load/store instruction is selected per bank. Since the address is required, it is imperative that a load/store instruction is split into two instructions: an address calculation instruction and a memory access instruction. A disadvantage is that more addition units are required which in turn increases the forwarding path time resulting in a higher clock cycle. Notice that neither buffers, nor a cross-bar is required between the load/store unit and the L1-cache. In fact the buffering is taken over by the instruction window. So an advantage of this approach is that the load/store-latency is deterministic (making abstraction of cache misses). At first sight, this seems like a scalable solution. Nothing is further from the truth; the cross-bar function should now be carried out by the selection logic, which selects an instruction for execution on a load/store unit. While in the traditional approach (Figure 2) the selection logic can be partitioned, so that only part of the instruction window is considered, in the apparent solution the entire (load/store) instruction window has to be considered. It was shown in [2] that selection logic is in the critical path of the core of the processor and that partitioning is required for future fast clocked designs.

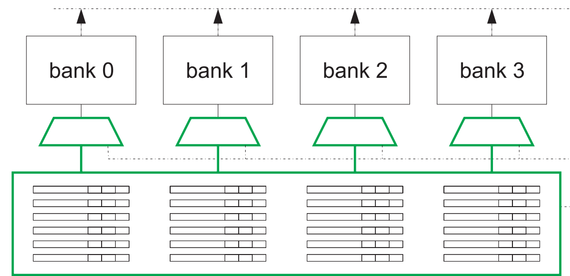


Figure 3. The apparent solution.

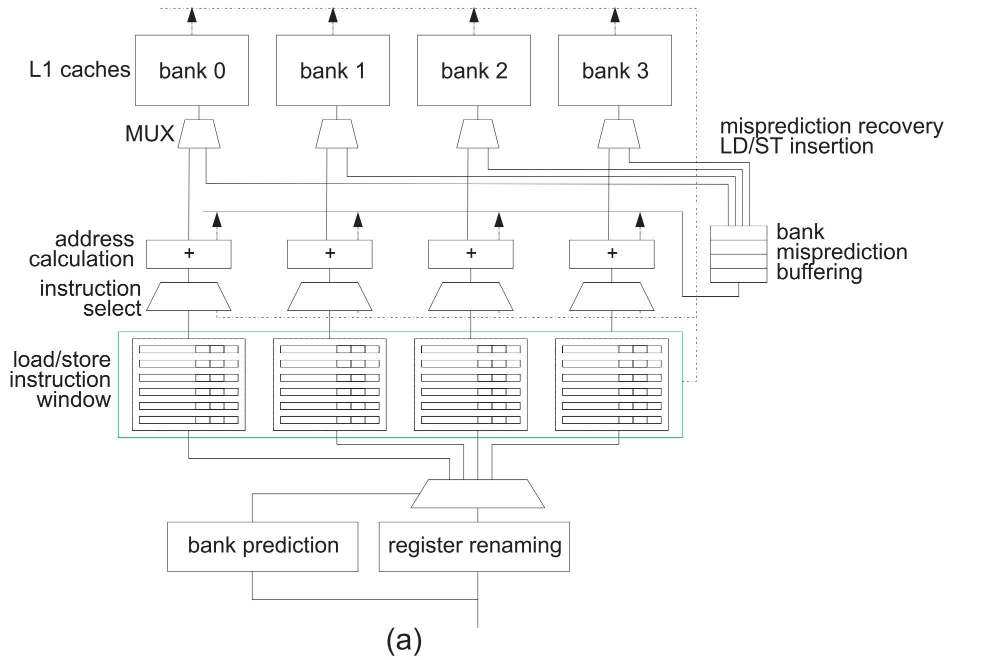
Another drawback of the apparent solution is the fact that the selection logic has to know the memory address for instruction selection. So the traditional technique of forwarding in dynamic processors, which inserts the values after the selection logic made a selection, will no longer be useful. Hence, every load/store could potentially take one extra cycle. More accurately, every load takes one extra cycle and every store that receives the address later than the value to be stored also takes an extra cycle (33 % of all stores for SPECint92).

2.3. Our solution

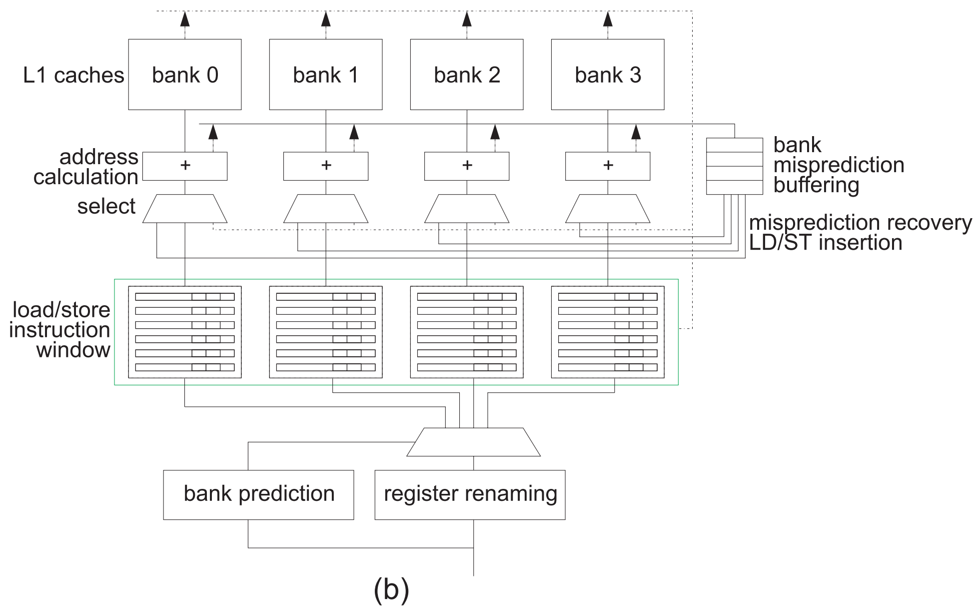
Thus it has become clear that we do not want the cross-bar in the critical path [2] which consists of instruction wake-up, selection, execution and result distribution. So why not shift the cross-bar one stage upstream, in front of the instruction window? This is the solution that we propose. Instead of putting the cross-bar in the issue logic or the execution logic, we put it in the less critical dispatch stage. While previously, the cross-bar latency was visible by every load/store instruction, in the new solution, only when there is a branch misprediction or an instruction cache miss, the extra cross-bar latency takes effect. However one difficulty remains; how can we decide which bank the instruction should go to without knowing the address? The answer is simple: prediction. This prediction has similarities to address prediction with the difference that we do not have to predict the whole address but rather the much smaller bank number. Another difference is that we have to make a prediction for *every* load/store instruction which is not required in the case of address prediction.

3. The prediction based implementation

We present two slightly different bank prediction based solutions in Figure 4. In both cases only a multiplexer is introduced in the critical path of the load/store unit. Scaling the solution to more banks does not lengthen this critical path. Only in the case of a bank misprediction, a longer latency is observed. Since bank prediction can be fairly accurate (87 %, see further), this latency has a small effect on performance. The difference between the two proposed solutions is the position where the mispredicted load/store is



(a)



(b)

Figure 4. Overview of our solutions.

inserted, at the end of the selection or, alternatively, after the address calculation. It is not clear yet which alternative is best; probably it is the insertion after the address calculation, so that this calculation is not repeated. On the other hand, it could prove easier to *reinsert* the instruction during selection i.e. reselect the instruction. A detailed hardware study should shed more light on what alternative is best.

Although the dispatching of the instructions into the predicted bank queue of the instruction window introduces some latency, the actual bank prediction itself takes place before dispatching and does not introduce extra latency. The prediction can be done in parallel with other stages like for example register renaming. Moreover, the extra dispatch latency is only visible in case of a branch misprediction or an instruction cache miss.

4. Bank prediction

So far we did not discuss the prediction strategy, nor the predictability of bank numbers. It is obvious that it is very similar to memory address prediction, since knowing the memory address implicates exact knowledge of the bank number. Predicting addresses has been investigated previously in the context of prefetching. The techniques used were computation based (e.g. stride prefetching [3]) or context based (e.g. Markov prefetching [4]), following the classification of [5]. In the latter work, the two prediction mechanisms, computation based in the form of stride prediction and context based in the form of a finite context method (FCM) were investigated for the more general case of value prediction. This study was performed for all kinds of instructions, not just for loads/stores, and showed good predictability. Hence, can we simply apply the same prediction schemes to bank prediction?

4.1. Discussion

In contrast to value prediction, by which 32-bit or 64-bit values are predicted, we are only concerned with $\log_2 \#banks$ bits that have to be predicted. The position of these bits is dependent on the way the banks are assigned to separate address spaces. Several conflicting factors influence this address space partitioning: (i) For one thing, it should be easy to replace a cache line. Due to this, a cache line is not spread over different banks but is contained in one single bank. If this were not the case a cache miss would affect all the banks. (ii) And another thing is that loads/stores executing in the same cycle should go to separate cache banks. Due to the spatial locality property of data accesses, the lowest bits should be used to address the banks. On the basis of these two arguments, a trade-off was made and the $\log_2 \#banks$ bits just next to the 5 bits that address the words in a 32-byte cache line are used as the bank number. Since the bank partitioning problem was not the focus of our research we made this reasonable 'ad hoc' decision. This address space partition is assumed in the remainder of this paper until stated otherwise.

4.1.1 Bank prediction against value prediction

Now, let us consider a stride based sequence of addresses:

8 16 24 32 40 48 56 64 72

Then the corresponding sequence of bank numbers (for a 4-bank cache) will be:

0 0 0 1 1 1 1 2 2

This example shows that releasing a stride based prediction algorithm on the bank number sequence, even though the address sequence is stride based, can be futile. This also follows from simulations. The poor predictability with the bank number based stride predictor is caused by the possible carry-over of intra-cache line bits to the bank number bits. These bits are not caught in a stride mechanism based on bank numbers. Hence, all the $5 + \log_2 \#banks$ least significant bits of a data address should be considered for stride prediction and these same bits should also be considered for the start address of the stride prediction scheme.
For the context based model, the situation is less clear and simulation results should tell whether the bank numbers are sufficient for a good prediction given a finite context method of a certain order. For an excellent description of the finite context method used in this study, we refer the reader to [5].

So a difference between address prediction and value prediction is that at most the $5 + \log_2 \#banks$ lower bits have to be stored instead of the whole data address or value. Address prediction also differs from bank prediction in that *every* load/store requires a prediction. Another significant distinction is that only $\#banks$ different constant values have to predicted. Because of this, even a random guess on unpredictable loads/stores, has a probability of $1/\#banks$ of being correct. For few banks, this can be a considerable percentage. It is also possible to take more educated 'random' guesses by limiting the set out of which you take a guess to a number of elements smaller than the number of banks, e.g., by considering only the 'even' bank numbers or more generally the $\bmod n + i$ bank numbers with $n, i < \#banks$ and constant. One last important distinction between address/value prediction and bank prediction is the higher static predictability.

4.1.2 Static prediction

While in the case of value prediction it seems very difficult if not impossible to predict the value statically, in the case of bank prediction this becomes easier. Consider for example references to the heap or the stack; these are not known statically. However, it is conceivable that the compiler is adapted to allocate the beginning of the stack frame of a function to a certain bank, so that the bank numbers of the accesses of the function to the stack are statically

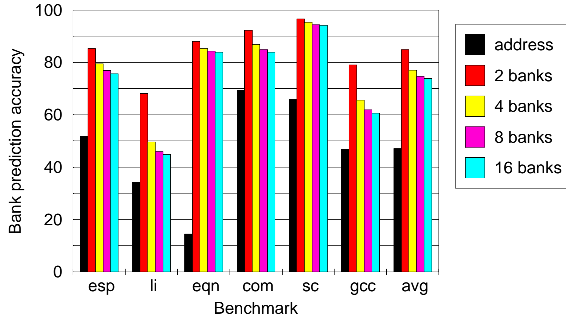


Figure 5. The prediction accuracy of last value prediction.

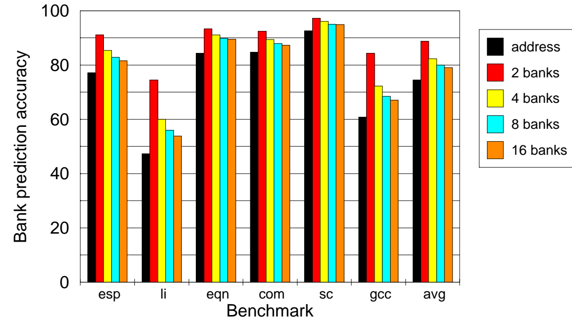


Figure 6. The prediction accuracy of stride prediction based on the lowest $5+\log_2 \#banks$ address bits.

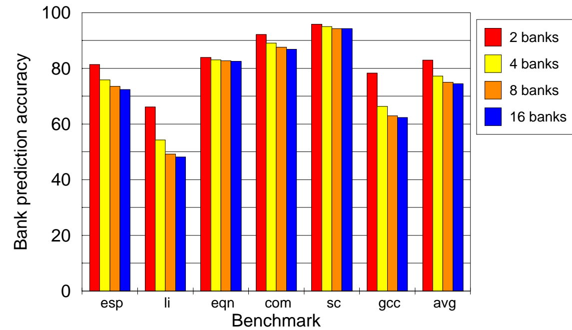


Figure 7. The prediction accuracy of stride prediction based on the bank number bits only.

predictable. Furthermore, knowledge of the bank numbers means knowledge of possible bank conflicts[1] so that better data allocation is possible. A similar situation arises for heap accesses; dynamic allocation routines in the library (e.g. malloc) could be adapted to use an extra parameter, the bank number. The allocated space would then start at the bank number so that a lot of heap accesses could be predicted. Here also, data allocation by the compiler could be optimized for both better bank usage and better predictability. Nevertheless, there should be no doubt that a lot of bank accesses will remain unpredictable at compile time.

4.2. Predictability

In this section we will take a closer look at bank predictability with computational based bank prediction, context based bank prediction and hybrid bank prediction techniques. Simulations results are presented first for infinite prediction tables and then for finite prediction tables. We also investigate the influence of compulsory misses, cold-start misses and conflict misses. All averages refer to arithmetic averages.

4.2.1 Simulation methodology

Bank predictability results were gathered through trace driven simulations. The traces were collected from SPEC integer benchmarks on a DEC 5000/125 station with a MIPS R4600 processor. The SPEC integer benchmarks have been compiled with the DEC `cc` compiler with the optimization flag set to `-O2`. More information on the traces is provided by Table 1.

4.2.2 Last value based prediction

The most simple predictor is the last value predictor. As the name states, this predictor simply assumes the last value as prediction. The accuracy of this predictor is fairly high, for two banks 85 % is attained, for 8 banks 75 % is attained, and for 16 banks still 74 % is attained (see Figure 5). For reference, address prediction is also plotted. These results show that bank prediction is much easier than the more general full address prediction. For example, eqntott has an address prediction accuracy of only 14 %, while the 8 banks prediction accuracy is as high as 75 %. So even a very simple prediction scheme with very little storage needs already attains reasonable bank prediction accuracies. These results are for infinite tables; further on finite storage space will be considered. However considering the few number of static loads/stores in traces with 50 million instructions (see Table 1) no big difference due to finite tables is anticipated. Only the conflict misses could still make a significant difference.

4.2.3 Stride based prediction

A more elaborate predictor is a stride based predictor. Again, we assumed infinite storage tables so that no aliasing, thus no conflict misses, occur. The stride predictor stores an actual stride value and a back-up stride value. The actual stride is used in a prediction. The back-up stride replaces the actual stride value when the back-up stride is encountered in two consecutive predictor invocations. Through the back-up stride mechanism, stride hysteresis is built in. We investigated two variations of stride based bank prediction:

[1] A bank conflict occurs when two instructions try to access the same bank in the same cycle.

program	input	dyn. instr count	LD/ST %	dyn. #LD/ST	stat. #LD/ST
espresso	mlp4.in	50 M	21.7 %	10.85 M	4722
xlisp	li-input.lsp	50 M	37.7 %	18.86 M	1819
eqntott	int_pri_3.eqn	50 M	25.8 %	12.88 M	795
compress	in	50 M	27.7 %	13.83 M	344
sc	load1	40 M	25.0 %	10.00 M	2419
cc1	tree.i	50 M	33.7 %	16.87 M	22121

Table 1. Information about the SPECint traces used in the simulations.

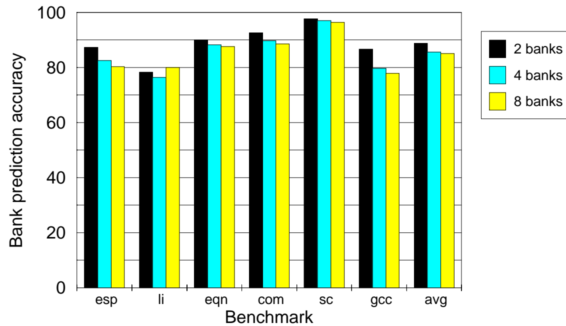


Figure 8. The prediction accuracy of the finite context method (FCM) predictor of order 3 with 4 bit counters for 2, 4 and 8 banks.

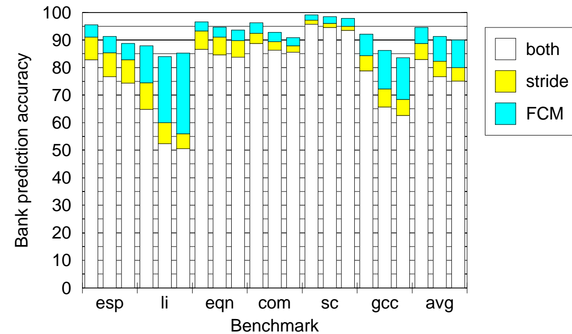


Figure 9. The prediction accuracy of the finite context method (FCM) predictor of order 3 with 4 bit counters.

(i) The first one uses the lowest $5 + \log_2 \#banks$ bits of the address. This mechanism performs significantly better (Figure 6) than last value prediction with an average of 80 % for 8 banks. (ii) The second stride prediction scheme uses only the $\log_2 \#banks$ bits next to the 5 intra-bank bits. The results are shown in Figure 7. An average bank prediction accuracy of 75 % for 8 banks was measured. From the latter results, and as expected, it is clear that all the lowest $5+\log_2 \#banks$ bits are required for good stride based bank prediction.

4.2.4 FCM based prediction

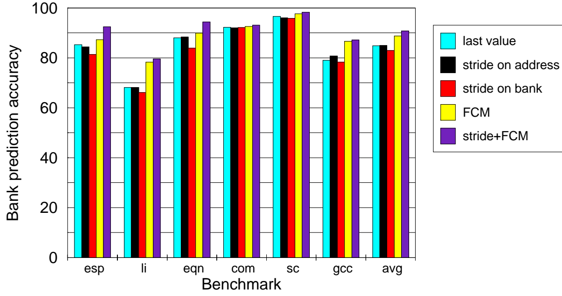
We investigated the context based prediction method called finite context method predictors (FCM) introduced by Sazeides *et al.* [5] for the more general case of value prediction but we applied the FCM to bank prediction. Here also infinite tables are assumed. For the context based prediction of order 3 (and only order 3) we get the bank prediction results of Figure 8. For 8 banks, a prediction accuracy of 84 % was measured. Other parameters used in these simulations are 4 bits for the FCM counters and subtraction of one from all counters in case a counter saturates.

4.2.5 Hybrid predictors

From the previous results it is clear that, on average, context based prediction results in better performance than stride prediction. However, context based prediction has a longer learning time and requires more information storage. So stride prediction should be used whenever applicable. Only when stride prediction performs badly, context prediction should be used. We will now investigate the possibilities of such a hybrid predictor.

The classification of the correct predictions in those correctly predicted by both stride and FCM, stride only, and FCM only (Figure 9) shows that most of the predictions can be correctly made by both prediction mechanisms. These results also show the maximum attainable prediction accuracy with a meta-predictor that chooses between the two mechanisms; i.e. a stride-FCM hybrid predictor with a perfect meta-predictor would give the results of Figure 9. Notice that especially xlisp and gcc leave a lot of room for improvement with new predictors. The programs xlisp and gcc are resp. pointer intensive and control flow intensive, implying possible directions to start looking for better predictors. New predictors could tackle the pointer problem through global correlation information, i.e. cross-instruction history. An example of a data value predictor based on cross-instruction information was presented in [6]. Control flow intensive programs could also be tackled with cross-instruction information. An alternative is the use of branch correlation information. We will neither introduce nor study such new predictors in this paper.

For the hybrid predictions, a meta-predictor i.e. a two bit saturating counter, is used to select between stride prediction and FCM prediction. The results (Figure 10) show that a bank prediction accuracy of 87 % is attainable for 8 banks. This is indeed less than the expected 90 % which can be derived from Figure 9. The discrepancy is caused by the fact that the meta-predictor is not an oracle. A comparison between the different predictors for the case of infinite tables



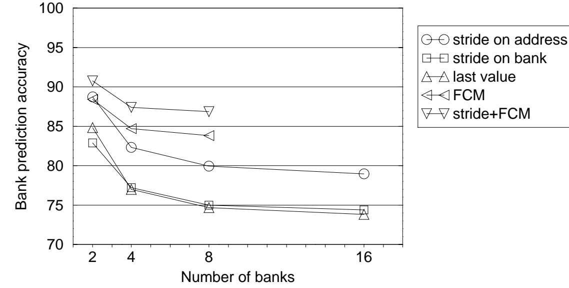


Figure 10. The prediction accuracies of the different predictors compared. Infinite tables are assumed. The left graph is for 2 banks.

	esp	li	eqn	comp	sc	cc1
compulsory misses	0.04%	0.01%	0.006%	0.002%	0.02%	0.13%

Table 2. The absolute percentage of compulsory misses.

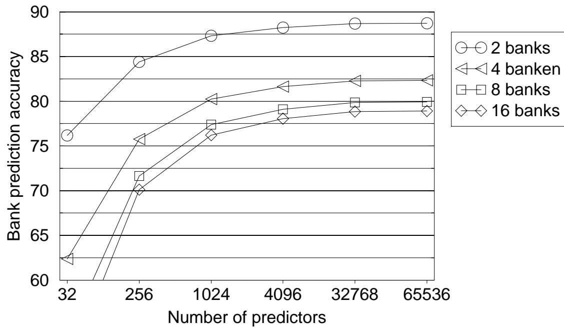


Figure 11. The stride prediction accuracy for finite direct mapped tables.

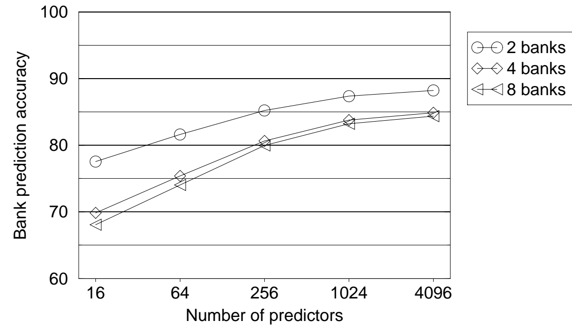


Figure 12. The predictability of the FCM scheme for finite direct mapped tables.

is presented in Figure 10.

From Table 1 the number of compulsory misses can be calculated (see Table 2). From these numbers we conclude that the contribution of compulsory misses is negligible.

Finite tables were also simulated. We did not use tags in these tables so aliasing can occur. The results for finite tables are shown in Figures 11, 12 and 13.

4.3. Division of the address space

In all previous simulations it was assumed that bit 5 (for 2 banks) or bits 5 and higher (for more than 2 banks) are predicted. This decision was made to keep the whole cache line in one bank so that in case of a cache miss, one bank only gets disturbed. However other choices of prediction bits are of course also possible. If you predict for example the least significant bits[2] (see Figure 14) then a higher predictability is observed. For two banks we even get prediction accuracies as high as 97 %.

The effect of the address space division on the prediction accuracy for 2 banks is plotted in Figures 15, 16, 17 for some selected programs.

[2]More accurately, the bits next to the intra-word addressing bits. The intra-word bits are not considered because all accesses are word aligned.

5. Processor performance

In the previous sections predictability was studied. This does not tell whether the bank prediction scheme is better than the traditional cross-bar approach or not. Nor is it clear what the cache performance or processor performance will be. We tackle these problems in a simplified way in the following sections. We identified two positive effects of our bank prediction solution: (i) a lower load/store latency and (ii) the deterministic character of the latency. The performance gain of the first effect will be estimated through a partial analytical model. The gain of the second effect is determined through simulations.

For the simulations in this section we used our own trace driven simulator hsim which models an out-of-order processor. The exact simulation parameters are given in Table 3. The same SPEC benchmarks traces as before were used.

5.1. The lower load/store latency

To evaluate the effect of the lower load/store latency, we use two metrics: the memory performance and the processor performance.

simulation configuration			
fetch bandwidth (instructions)	16	L1 inst cache size	64Kb
issue bandwidth (instructions)	12	L1 inst cache associativity	1
#load/store units	2 or 4	L1 data cache size	64Kb
loads/stores ordered	no	L1 data cache associativity	1
instruction window size (instructions)	160	L2 cache size	2048Kb
reorder bandwidth (instructions)	40 (≈infinite)	L2 cache associativity	2
#pipeline stages in front of window	5	L2 cache hit latency (cycles)	7
load/store latency	2 cycles, pipelined	memory access latency first word (cycles)	70
integer unit latency	1 cycle	memory access latency first to next word (cycles)	5
branch prediction accuracy	96%	memory access bus width (bytes)	8

Table 3. The processor configuration used in the simulations.

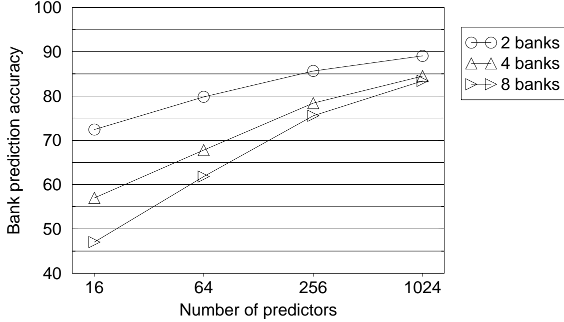


Figure 13. The predictability of the hybrid scheme for finite direct mapped tables.

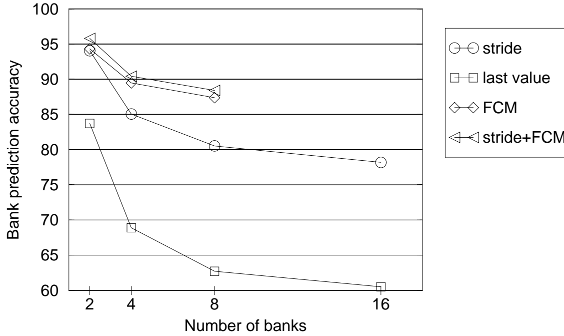


Figure 14. Prediction accuracies for lowest order bits interleaved banks.

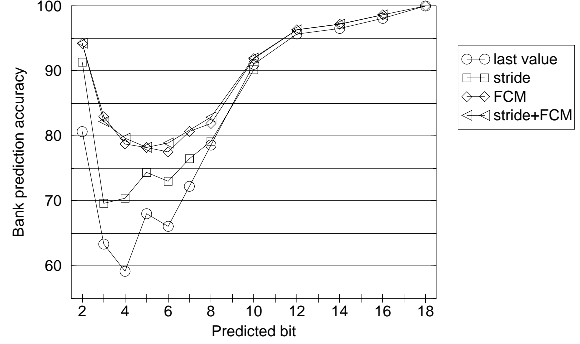


Figure 15. Prediction accuracies of xlisp for different interleaving bits (2 banks).

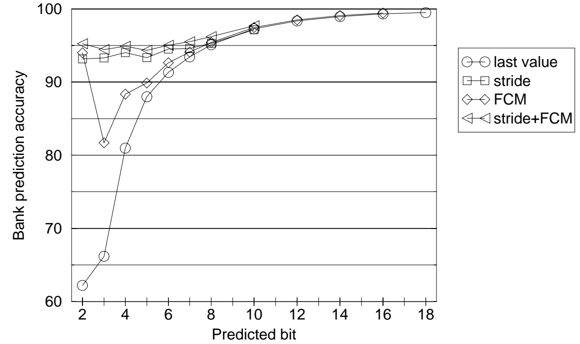


Figure 16. Prediction accuracies of eqntott for different interleaving bits (2 banks).

5.1.1 Memory performance

An often used and simple memory performance measure is the average memory latency [7]. We define a similar quantity for the L1-cache accesses that we study: the average L1-cache hit latency. When l represents the L1-access latency (including address calculation) in clock cycles, p the bank prediction accuracy and if we assume that a bank misprediction introduces an extra latency of one cycle then the average L1-cache hit latency (including address calculation) expressed in clock cycles is:

$$p \times l + (1 - p) \times (l + 1) = l + 1 - p$$

So the bank misprediction probability equals the extra time an L1-cache hit takes compared to perfect prediction. For the cross-bar solution, given an extra latency of x (expressed in clock cycles as time unit), the L1-cache hit latency is:

$$l + x$$

Note that congestion in the cross-bar is not accounted for, since this same congestion was not accounted for in the bank prediction model. It follows from these equations that the difference in memory latency is:

$$1 - p - x$$

This difference is the latency increase when going from the cross-bar to the bank prediction technique. If bank prediction is perfect ($p = 1$) then the bank prediction technique is always better.

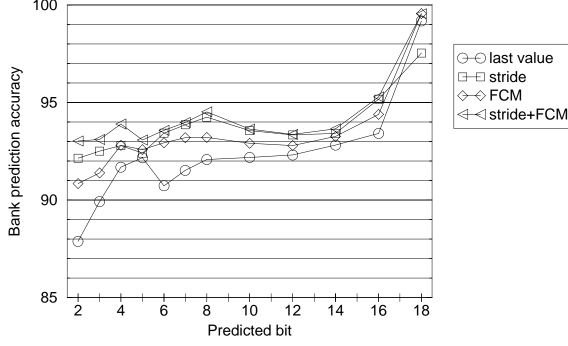


Figure 17. Prediction accuracies of compress for different interleaving bits (2 banks).

Remark that this estimation is only a first order estimation. Also, the ultimate metric is processor performance so we will now estimate this number.

5.1.2 Processor performance

We assume that the **cross-bar solution** has only an effect on the latency of loads/stores, but neither on the latency of other functional units, nor on the repetition rate of any unit. Then the relative performance change, compared to a zero latency cross-bar is (x is as before the extra latency introduced by cross-bar and buffer-management):

$$\Delta_l IPC(l) \times x$$

with

$$\Delta_l IPC(l) = \frac{IPC(l) - IPC(l+1)}{IPC(l)}$$

So $\Delta_l IPC(l)$ expresses the sensitivity of the IPC to an increase of the load/store latency with one clock cycle. This quantity can be determined by simulations.

The **bank prediction** technique not only affects the average L1-hit latency but also the latency of the dispatch stage.

For the bank prediction technique we assume a bank misprediction penalty of one cycle for the load/store latency. Only a fraction $1-p$ of the load/stores experience this extra latency, so the performance decrease caused by mispredictions is given by:

$$\Delta_l IPC(l) \times (1-p)$$

The latency of the dispatch stage also increases slightly, because of the cross-bar, with a latency x. Only when there is a branch misprediction (and maybe for an L1 instruction cache miss) this latency affects performance. This effect of branch mispredictions and instruction cache misses on performance is captured by:

$$\Delta_s IPC(s) \times x$$

with s the number of pipeline stages upstream of the instruction window and

$$\Delta_s IPC(s) = \frac{IPC(s) - IPC(s+1)}{IPC(s)}$$

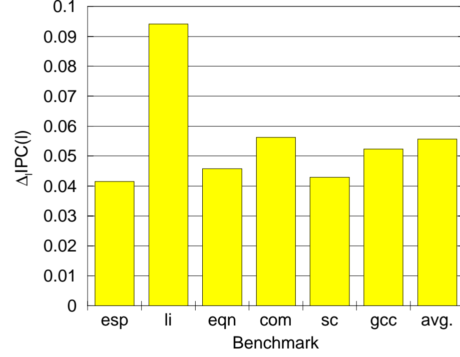


Figure 18. The sensitivity of the performance to an extra cycle of load/store latency.

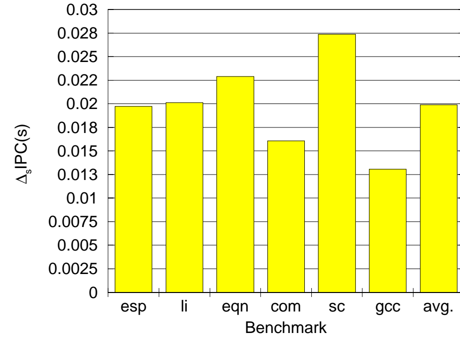


Figure 19. The sensitivity of the performance to the latency of an extra pipeline stage upstream of the instruction window.

So it follows that the performance gain of the bank prediction technique over the cross-bar solution is given by:

$$(\Delta_l IPC(l) - \Delta_s IPC(s)) \times x - \Delta_l IPC(l) \times (1-p) \quad (1)$$

Using simulation determined sensitivities (see Figures 18 and 19) and simply assuming perfect bank prediction, equation (1) is reduced to $3.8\% \times x$. So even if we can eliminate half a clock cyle by removing the cross-bar and the buffer management, the performance increase is limited to a mere 1.9 %. This increase is not worth the extra hardware needed for bank prediction. However, this shows only the effect of the reduced load/store latency. How about the performance contribution by the more deterministic character of the load/store latency?

5.2. Deterministic latency

In an out-of-order processor, the deterministicity of an instruction's latency is very important. To enable back-to-back execution of instructions, instructions have to be issued back-to-back. This implies that the issue stage knows the latency of the instruction. Remark that this is not the case when a cache miss, a bank conflict (in the cross-bar solution) or a bank misprediction occurs. If this latency was underestimated then dependent instructions could have been is-

sued which should not have been issued. These instructions can not execute since they will not receive all their operands through forwarding; a stall occurs.

In the Alpha 21264 [1], the problem of a non-deterministic latency is (re)covered (from) by flushing the last issued instructions and reissuing them later instead of stalling all pipelines. This operation takes two cycles. To incur this penalty the least possible, a hit/miss prediction mechanism was implemented. In case of a prediction of a miss, the dependent instructions are not issued until the result is really available.

In our simulations we assume a similar mechanism as in the Alpha 21264. Bank conflicts in the cross-bar technique and bank mispredictions in the bank prediction technique incur a two cycle minirestart penalty. Consumers of the conflicting or mispredicted instructions should wait until the result from that instruction actually arrives; no forwarding is possible. Because the forwarding is disabled, an extra cycle (on top of the two minirestart cycles) compared to a back-to-back execution is introduced. If there is no bank conflict or no misprediction then instructions execute normally, back-to-back.

Simulations with a finite hybrid predictor yields the results of Figures 20 and 21 for respectively 2 and 4 banks. We also simulated perfect bank prediction so as to have an idea on the maximum attainable performance with better prediction. With perfect bank prediction one could gain on average 32 % (resp. 33 %) for 2 banks (resp. 4 banks) with the bank prediction technique over the cross-bar technique. With real prediction, the performance gain is 16 % for 2 banks and 9 % for 4 banks. Remark that xlisp and gcc perform better with the cross-bar than with real bank prediction. Since this effect does not occur with perfect prediction, the cause must be the low bank predictability of both. Indeed these two programs have the lowest prediction accuracy. To put it another way, there are more bank mispredictions than bank conflicts. This conclusion can be drawn because the penalties for a bank conflict and a bank misprediction are equal.

One could think of other more complicated, lower penalty recovery mechanisms than the Alpha 21264 implementation to handle the non-deterministic latency in the case of bank conflicts or bank mispredictions. However space limitations prevent a discussion of these techniques. We still mention that as a result of the lower penalty a smaller performance gain can be expected. Nevertheless, the (in most cases) lower number of bank misprediction over the number of bank conflicts still gives the performance edge to the bank prediction technique.

6. Bank and address calculations

The goal of the bank prediction technique is to provide a low latency and deterministic latency for a load/store instruction. In this we succeeded for a power of two number of banks. But for a number of banks 2^n+i with $1 < i < 2^n-1$, determining the bank number and the intra-bank address is a more complex operation than shifting; a full division is required taking a long latency. Here again the question arises, which low latency solutions do exist?

One solution is to divide the address[3] by 2^{n+1} and to map the set of 2^{n+1} remainders to $2^n + i$ banks. Consider for example 3 banks. Division by 2^2 results in the remainders 0,1,2,3. One can then for example map 0,1,2,3 to banks 0,1,2,0. A major drawback is that some banks, bank 0 in the example, receive double as much traffic. So this solution is badly balanced. The compiler could help a little bit in distributing the traffic more evenly over the banks by allocating the data in a special way. How good a compiler is able to perform this job deserves further research. But we believe that this solution will result in sparser data allocation and more irregular address calculation than the solution that we propose now.

A low latency solution with balanced traffic is possible with the use of quotient-remainder address representations. In the address calculation, quotient and remainder—the divisor is $\#banks$—are kept separately and every addition or subtraction is performed on the quotient and remainder separately, with possible carries propagated. This way, the division never occurs at runtime, but is executed once, at compile time only. Then the address calculation in the load/store unit only consists of simple additions and subtractions, no complicated long latency divisions are required. Furthermore, since only additions and subtractions are required, the low-latency combined addition-decoding logic approach proposed in [8] can be used.

6.1. Related work

The importance of the load latency was demonstrated in [9]. Hence, several low latency cache access approaches—complimentary to our approach—have been proposed. Austin *et al.* [10] speed up the address calculation by speculatively performing an OR operation instead of a full addition. They performed a performance study assuming the address calculation takes a full cycle. However, as pointed out in [8], in modern processors the address calculation only takes a small fraction of a cycle. In [8] an even better address calculation scheme was demonstrated. The authors show that it is possible to combine the addition logic with the address decode logic into a low latency circuit. Unfortunately, they only discuss the decrease in load latency not the performance improvement.

Concerning prediction, there are two kind of studies, those on prefetching and those on value prediction, that have some resemblance to our approach. Like prefetching, we try to predict, albeit only part of, the data access address. It has

[3] With 'address' in this and the next paragraph, the address bits without the intra-bank bits is meant.

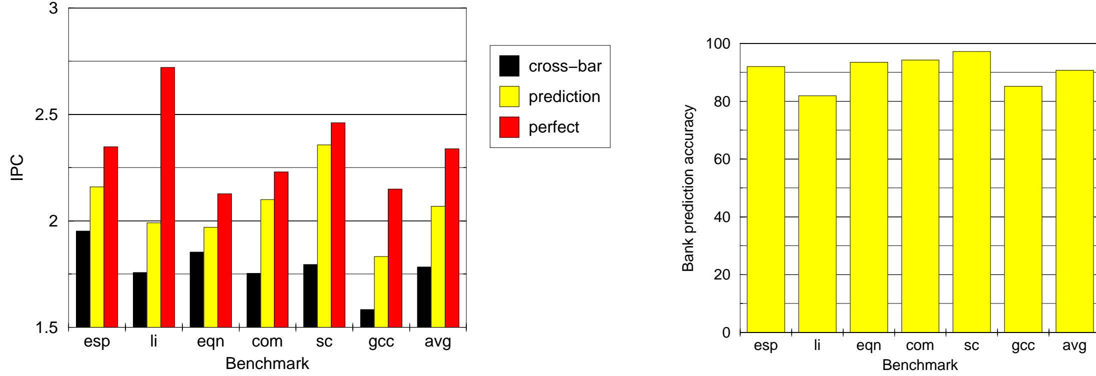


Figure 20. The IPC (left) for the cross-bar solution, for perfect bank prediction and for real bank prediction. Two banks are assumed. Bank prediction accuracies are shown on the right. Bit 5 is predicted.

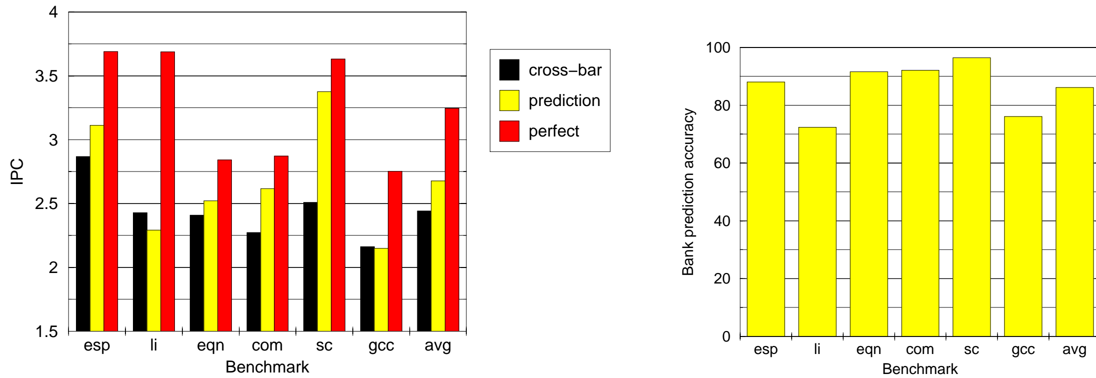


Figure 21. The IPC (left) for the cross-bar solution, for perfect bank prediction and for real bank prediction. Four banks are assumed. Bank prediction accuracies are shown on the right. Bits 5 and 6 are predicted.

already been demonstrated in prefetching that stride-based address prediction [3] and context based (Markov chain) address prediction [4] are useful prediction algorithms. A notable difference is that we predict only part of the address and we always try to predict the address of every load/store, while in prefetching the purpose is to predict cache misses only. For our prediction schemes we depended strongly on those investigated by Sazeides and Smith in [5]. These mechanisms seemed the most applicable to our problem. One important difference is that they investigated the values loaded by loads/stores (and the values produced by all kind of instructions classified according to instruction type). We studied the addresses *generated* by loads/stores which are inherently more predictable than the values stored at these addresses. Furthermore, we only have to predict *some* bits which further increases the predictability and reduces the information that has to be stored for the prediction considerably.

For set associative caches, Calder *et al.* proposed in [11] prediction of the set so that the latency is, in case of a correct prediction, that of a direct mapped cache. They predict the set that will be accessed in a two-way set associative cache. Notice that the set for a given memory location can change in time, and that they predict something fundamentally different from the bank number. Consider for example the excellent stride prediction technique for bank prediction; for set prediction this makes no sense at all. Because the prediction of a set is so fundamentally different, they use other prediction techniques than the ones used in this paper.

Around the same time that we introduced bank prediction, Yoaz *et al.* [12] also reported preliminary work on bank prediction. They investigated bank prediction mainly by using branch predictor algorithms to predict one out of 2 banks. A performance evaluation was not presented. It is proposed that loads/stores are duplicated when the bank prediction confidence is low. It is also argued that in this way the scheduling is improved. But this method of duplication does not avoid the latency non-determinism unless all duplicated instructions are issued at the same time.

In another recent investigation [13] on cache banks the

usage of semantic caches was introduced; a load/store is determined to be a stack access or a non-stack (e.g. heap) access and one cache bank holds stack items while the other cache bank holds non-stack items. Whether a load/store instance is a stack access or not can be determined with a prediction accuracy of 99.9% [13]. Remark that a wrong classification of only one access can leave the data in or fetch the data from the wrong bank, with data inconsistencies as a result. One flaw of this approach is that there is no possibility to check for incorrect (semantic) predictions so one has to be sure to have a perfect predictor. Their technique is only applicable to two banks but can be combined with our bank prediction techniques for more banks.

7. Conclusions

Traditionally, to connect several load/store units to several cache banks, a cross-bar, buffers and buffer management are required. These introduce latency in the critical path of load/store execution. It is observed that this latency can be eliminated by connecting every load/store unit to its own cache bank. To enable this simple interconnection scheme we had to introduce a new technique: bank prediction.

Furthermore, the non-determinism of the load/store latency caused by bank conflicts in the traditional cross-bar interconnect is now eliminated. In the case of a correct bank prediction, the load/store latency is low and deterministic for L1-cache hits. Only in the case of a bank misprediction the load/store latency is higher (mostly one cycle) and there is non-determinism in the load/store latency. So it is important to have a high bank prediction accuracy.

The bank prediction accuracy was studied extensively and it was shown that average prediction accuracies of 90 % for two banks, 85 % for 4 banks and 84 % for 8 banks are feasible for finite tables. These prediction accuracies are resp. for the prediction of bit(s) 5, 5-6 and 5-7. This choice was made to keep the whole cache line into one bank. For two banks, we also investigated the predictability of other bits. Predicting the least significant bit increases the prediction accuracy to 94 % and higher.

The possible performance gain with bank prediction over the traditional cross-bar solution is estimated to be lower than 1.9 % when the shorter latency is considered only.

However if the latency non-determinism is considered and handled as in the Alpha 21264 (with a minirestart) then the performance to be gained is as high as 32 % for 2 or 4 banks with perfect branch prediction. With realistic bank prediction the performance increases by 16 % for 2 banks and by 9 % for 4 banks. These results are for the prediction of bit 5 resp. bits 5-6.

References

[1] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE MICRO*, 19(2):24–36, April 1999.

[2] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.

[3] T. F. Chen and J. L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of ASPLOS-V*, pages 51–61, October 1992.

[4] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *The 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[5] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[6] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. Correlated load-address predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[8] William L. Lynch, Gary Lauterbach, and Joseph I. Chamdani. Low load latency through sum-addressed memory. In *The 25th Annual International Symposium on Computer Architecture*, pages 369–377, June 1998.

[9] Kunle Olukotun, Trevor Mudge, and Richard Brown. Performance optimization of pipelined primary caches. In *The 19th Annual International Symposium on Computer Architecture*, pages 181–190, June 1992.

[10] Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi. Streamlining data cache access with fast address calculation. In *The 22nd Annual International Symposium on Computer Architecture*, pages 369–380, June 1995.

[11] Brad Calder, Dirk Grunwald, and Joel Emer. Predictive sequential associative cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, 1996.

[12] Adi Yoaz, Erez Mattan, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.

[13] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 100–110, May 1999.