On Reconfigurable On-chip Data Caches

Fredrik Dahlgren and Per Stenström
Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 Lund, Sweden

Abstract

Cache memory has shown to be the most important technique to bridge the gap between the processor speed and the memory access time. The advent of high-speed RISC and superscalar processors, however, calls for small on-chip data caches. Due to physical limitations, these should be simply designed and yet yield good performance.

In this paper, we present new cache architectures that address the problems of conflict misses and non-optimal line sizes in the context of direct-mapped caches. Our cache architectures can be reconfigured by software in a way that matches the reference pattern for array data structures. We show that the implementation cost of the reconfiguration capability is neglectable. We also show simulation results that demonstrate significant performance improvements for both methods.

1 Introduction

Historically, there has been a discrepancy between the speed of the CPU and the memory system. With the advent of high-speed microprocessors, it has been increasingly important to address a number of issues in the design of high-performance memory systems.

Cache memories are used to increase the speed of the memory system, and by introducing on-chip caches, it was possible to achieve a best case performance much higher than ever before. But due to physical limitations, on-chip caches are very small, and their average performance are thus clearly limited. An early evaluation of on-chip caches can be found in [HS84].

A distinction is often made between *instruction caches*, only containing instructions, and *data caches* only containing data, see e.g. [Smi82]. It is shown possible to achieve good performance for instruction caches, either by additional hardware, see e.g. [FP89], or by an optimizing compiler allocating the code in a fashion suitable for the cache mapping in order to reduce the number of conflict misses, see e.g. [HC89] and [SG85].

Small data caches, on the other hand, are still suffering from a high miss ratio. Because of the implementation complexity and latency of a highly associative cache, it would be preferable to use a direct-mapped cache, if sufficient methods to improve the performance of such are proven applicable, see e.g. [Smi82].

In this paper, we present two new cache architectures that both can be reconfigured in a way that matches the reference pattern of array data structures.

The first cache architecture addresses the problem of conflict misses in direct-mapped caches. We propose an enhanced cache architecture, *the virtual cache unit technique*, which gives an opportunity to significantly reduce the mapping conflicts in a direct mapped on-chip data cache to an insignificant hardware cost.

The second cache architecture addresses the problem of non-optimal line (block) size for a given data structure (see e.g. Smith [Smi87]). We show that the optimal line size depends heavily on the workload and data allocation. The cache architecture can associate different line sizes to different data structures on a per-page basis.

Although the methods can be used orthogonally, the common properties they have are that they are examples of reconfigurable caches, they can be implemented to a neglectable hardware cost, and most importantly, they provide significant performance improvements. This paper covers all these aspects according to the following outline: The methods are presented in section 2. Section 3 explores the implementation of both techniques. Simulation results that confirm

the performance improvements are shown in section 4. In section 5, finally, we discuss the results obtained.

2 Methods to reduce data cache misses

Misses in a cache, on an item previously referenced, can be divided into three groups due to what caused them [HP90]. *Compulsory* misses are due to the first access to a line, which is not in the cache. These misses are also called *cold start misses* or *first reference misses*. *Conflict* misses are caused by the mapping-effect, e.g. in a direct-mapped cache where two items mapped to the same set never can exist in the cache at the same time. *Capacity* misses, or size misses, are caused by the fact that the cache is not infinitely large. Large line size means a high degree of prefetching into the cache due to the principle of reference locality, but this also means a high degree of conflict misses.

In the rest of this chapter, we attack the problems of conflict misses and the impact of the line size on the hit ratio. In section 2.1, a method to reduce the number of conflict misses in direct mapped data caches are presented. The method is called the *virtual cache unit technique*. In section 2.2, the principles of a variable line size cache is presented.

2.1 The virtual cache unit technique

One of the main reasons for the low hit-ratio in small direct mapped data caches are due to mapping conflicts between data in loops with many iterations. If two items, mapped to the same set, are referenced within a loop, there will always be a miss on both of them in every iteration of the loop. This is referred to as *bumps*. In [BS88], Breternitz and Shen discuss this problem as an extension to their contributions.

It is desirable to map data arrays in such a fashion, that the references to elements in different arrays accessed within an iteration of a loop are never mapped onto the same set in the direct mapped cache. This is trivial if the indexing pattern are exactly the same for the arrays, but problems occur if the indexing pattern between different arrays accessed within a loop are not the same. In this case, it is preferable to make sure that these arrays are never mapped onto the same set.

We also have a problem with non-array elements, if the loop traverse a long array. The problem is that if the arrays are larger than the cache, the indexing through the loop will affect every set in the cache at least once, and thus bumps will occur between the vectors and the non-vectorized variables. Here, it is preferable to make sure that the arrays are never mapped onto the same set as the non-array elements.

In order to reach an acceptable hit ratio for a direct mapped data cache when iterating in a loop, the two problems mentioned above must be taken care of.

An enhanced cache architecture is proposed, which gives rise to a solution to these problems. The technique is chosen to be called the *virtual cache unit technique*, because the cache organization is regarded differently for different data. A part of the memory area is divided into a number of memory subareas, see figure 1. In the following, the chosen part of the memory area begins at address 0, which is by no mean a requirement for this method.

The cache is regarded as consisting of an equal number of units, *cache units*, all having the same size. A cache unit works like a direct-mapped cache for its corresponding memory subarea. A large array allocated in this subarea can never be mapped into a set outside its cache unit, see figure 2. For the memory area outside the special subareas, the cache is regarded as one normal direct-mapped cache, and the different cache units are not visual. This is why the method is referred to as the *virtual cache unit technique*.

If the data of a program is allocated in the main area, i.e. outside the special subareas, the behavior of the cache is exactly as normal. The problem of bumps between an array and non-array elements can now be solved in a way according to figure 3. In this example we assume 2 cache units, and the arrays are allocated into the first subarea, while the non-array variables are allocated into the other. If the indexing pattern for the two arrays are the same, they are offset in a way that no bumps between them can occur. As can be seen from figure 3, no bumps occur between the non-array variables and any other element.

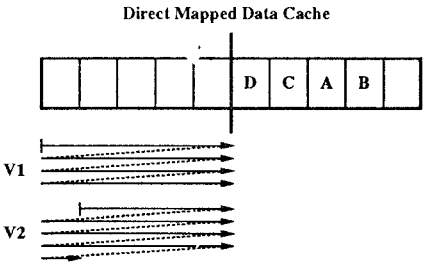

Figure 3: Using the enhanced cache architecture to avoid bumps

If the loops access two arrays, but there is no obvious connection between the indexing pattern for these arrays, they can be isolated from each other in different cache units, see figure 4. Bumps between the arrays can be avoided.

By allowing fine-granularity for the decomposition into cache units, the compiler has the possibility to decide where in the cache different data should reside, and there are no problems with array variables.

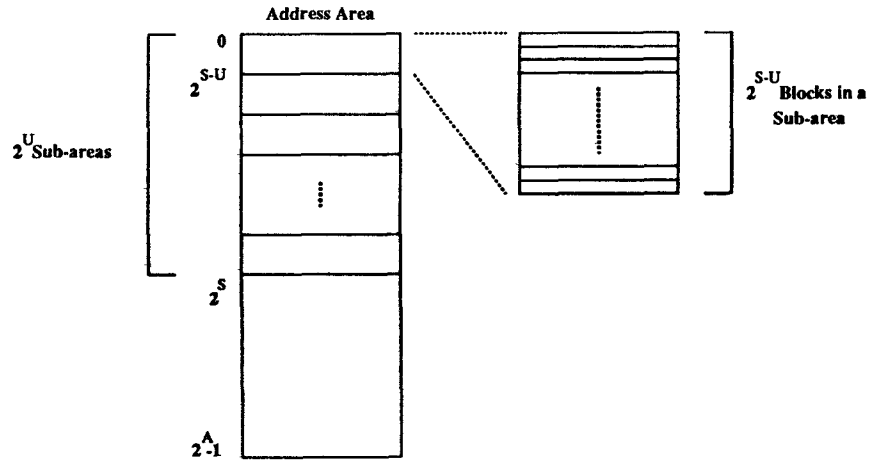

Figure 1: The dividing of the memory area into the special sub-areas

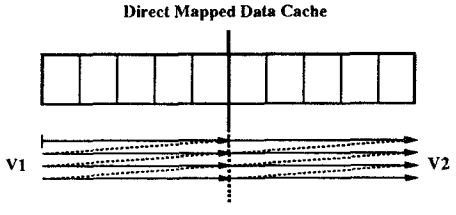

Figure 4: Using the enhanced cache architecture to avoid bumps between two non-predictably indexed arrays

In contrast to pure compiler methods, the virtual cache unit technique requires additional hardware, which is why we are talking about an enhanced cache architecture. The principles for the address-mapping, and thus the additional hardware required, are followed below.

The offset address field is not concerned, which is why the rest of this section only deals with line-addresses and the *tag*-field and *set*-field of the address.

Assumptions:

- 2^A blocks in the address space
- 2^B sets in the direct-mapped cache
- 2^S blocks in the address space for the special subareas
- 2^U cache units in the direct-mapped cache

The number of bits in the line address is thus A bits (bits 1 to A, where bit 1 is the least significant bit), distributed into B bits in the *set*-field and *A-B* bits in the *tag*-field. The address is within a special sub-area, if the bits *S+1* to A are all zero, see figure 1. In that case, the situation becomes the following:

Since the bits $1 \ldots S-U$ selects a certain line within a subarea, see figure 1, bits *S-U+1* to S determines which cache unit to be used. The 2^B sets in the cache are divided into 2^U equal-sized groups, which gives us that each of these groups contains 2^{B-U} sets. Bits 1 to *B-U* determine which set within the group should be pointed out, and the new set-field for the direct-mapped cache becomes the concatenation of the bit-fields $1 \ldots B-U$ and $S-U+1 \ldots S$. The bits $B-U+1 \ldots B$ must be included in the tag-field, as well as the bits $S-U+1 \ldots S$. Our new tag-field becames $B-U+1 \ldots A$. This is illustrated by figure 5 below. Both the C-field and the B-field contain U bits. The tag-field in the direct-mapped cache becomes U bits wider.

In section 3, we will show an implementation of this mapping strategy. In particular, we will demonstrate that the implementation cost is neglectable as compared to the cost of a direct-mapped cache. In section 5, we'll confirm that significant performance improvements can be obtained. We now pay our attention to variable-sized line caches.

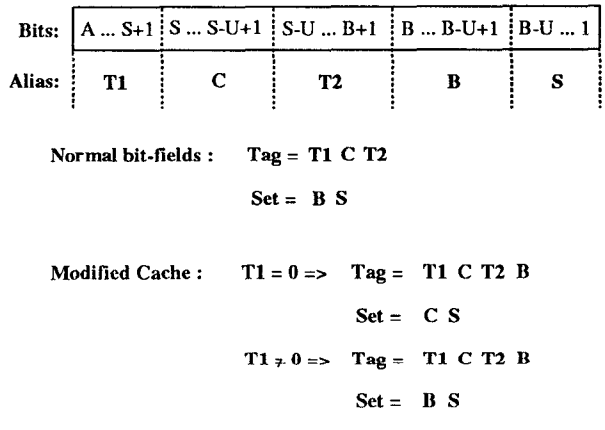

Figure 5: The address fields of a cache line

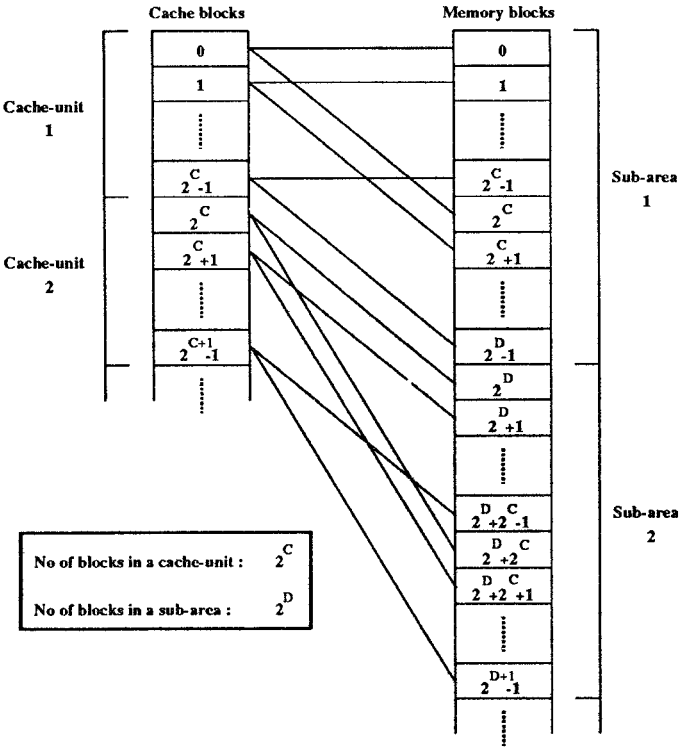

Figure 2: Cache unit mapping to the sub-areas

2.2 Variable line (block) size

One of the most important parameters of a cache organization is the line size [Smi87]. It has been observed that, for a given cache size, the optimal line size depends heavily on the workload and data allocation, which will be described in the result section. Intuitively, one realize that if a number of arrays are traversed in index order, and no bumping occur between the arrays, a high degree of prefetching is desirable. This means that the hit-ratio is increased with larger line sizes up to the point where the number of different lines referred to exceeds the number of lines in the cache. In such a case, we want a large line size. On the other hand, in a multiprocessor application with a high degree of parallelism, the performance degradation due to false sharing increases with larger line sizes, and it is definitely desirable with a small line size. This was observed by Eggers and Katz [EK89].

Assume the following case as an example. In figure 6 below, we have two programs for a multiprocessor, **P1** and **P2**. In **P1** the processors are updating elements in a shared array, and the elements are written more than once, while **P2** is traversing arrays which are local to the processors using them. In **P1**, with a small line size (almost no false sharing) we reach a high hit-ratio because the elements in the array are accessed more than once, but after point **A** the degree of false sharing increases. In **P2**, we don't have the problem of false sharing, so larger line sizes means a higher degree of prefetching, and therefor higher hit-ratio. After the point **B**, the number of lines becomes so small that mapping conflicts between different data variables increases dramatically.

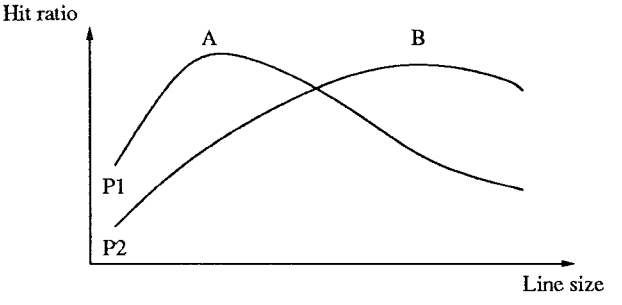

Figure 6: An example of the hit-ratio as a function of the line size for two different programs.

We propose a cache architecture with a variable line size. By using knowledge about program behavior, the optimal line size can be chosen, which give rises to significant performance improvements.

The smallest line possible is called a *basic line*, and the other line sizes possible are multiples of the size of this basic line. The basic cache architecture is similar to a normal direct-mapped cache with the basic line size. When a cache-miss occurs, more than one line might be read into the cache.

For example, if two basic lines are read into the cache each time a cache miss occurs, the behavior is the same as a direct mapped cache with a line size of twice the basic line. The only difference is that the two new cache lines are always consecutive *beginning* with the one on which the miss occurred. The case when the interesting part of the line is the latter of the new cache lines never occurs.

3 Hardware implementations

In this section, we will show an implementation of the virtual cache unit architecture, and the variable line size cache. In particular, we will demonstrate that the implementation cost is neglectable as compared to the cost of conventional direct-mapped caches. In section 5, we'll confirm that significant performance improvements can be obtained.

3.1 The virtual cache unit technique

Figure 5 shows the principles of the technique. The tags in the new cache will be U bits wider, and the **T1**-field will determine the set, but the rest is exactly the same as a normal direct-mapped cache. The determination of the set is done with a multiplexor, and the control signal of this is the test **T1** = *0*. Figure 7 shows the hardware needed, and the extra hardware besides the longer tags is a multiplexor.

A 2-way multiplexor of U bits can be implemented densely in CMOS by $2U$ N-transistors and $2U$ P-transistors forming $2U$ pass transistors [PE88]. Since $U = \log_2($ number of cache units) the cost of the multiplexor is neglectable for any reasonable size of the cache. For instance, 4 cache units yields a multiplexor containing 8 transistors for the pass transistor array.

The bit overhead in the tag store is also neglectable as shown in table 1. Here we assume $A = 32$ address bits, a direct-mapped cache of 2^B bytes, and various number of cache units 2^U. The overhead is given by $U/(A - B)$. The bit overhead is acceptable as long as the number of cache units is in the order of 10. However, the overhead grows logarithmically to the number of cache units.

It is very important that the extra hardware does not introduce performance degradation for every access, since the comparisons are always made. For physical addressed caches, this can be done without any performance loss at all. This is done by letting the set-determination paths through the multiplexor work in parallel with the address translation from virtual to physical addresses, see figure 8. If the page-size is not smaller than *No of sets* $\times$ *Line size*, i.e. not smaller than the direct mapped cache, the address fields **B** and **S** in figure 5 are not included in the address translation. The field **T1** in the virtual address ($\mathbf{T1}_V$) determines if the

No of cache units	Size of the cache	Overhead
2	1 Kbyte	4.54%
2	16 Kbyte	5.55%
4	1 Kbyte	9.09%
4	16 Kbyte	11.11%
8	1 Kbyte	13.64%
8	16 Kbyte	16.67%

Table 1: The bit overhead in the tag store.

virtual cache units are to be used, or if the cache is addressed as normal. In the case of using cache units, the field C in the virtual address (C_V) determines which cache unit to use.

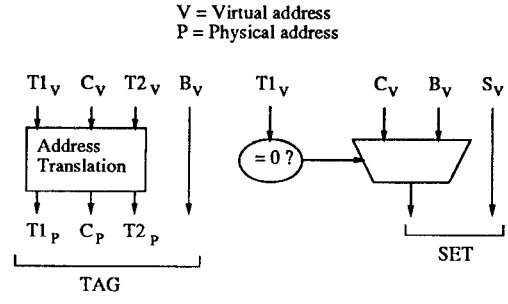

Figure 8: Parallelization of the set-determination and the address translation.

If the virtual cache unit technique is used for an n-way associative cache, the set-determination can work in parallel if $n \times$ *Page size* $\geq$ *No of sets* $\times$ *Line size*. For the processors i486 [Int90a] and i860 [Int90b], which both have a page size of 4Kbyte, this means that our enhanced cache do not have any performance degradation due to extra hardware for direct mapped on-chip data caches not larger than 4Kbyte, and for 2-set associative caches not larger than 8Kbyte.

3.2 The variable line size cache

An implementation is shown in figure 9. The address translation and direct-mapped cache are exactly as normal. The replacement controller will be slightly modified compared to normal. There are two new modules, the *line size table*, LST, and the *address incrementer*, AI. The LST contains the number of basic lines that should be fetched in the case of a cache miss. The LST could consist of just one entry which leads to the same number of basic lines for the whole program (except when explicitly changed within the program), or consist of a number of entries giving rise to different line sizes for different parts of the virtual address space.

The LST works in parallel with the address translation on every access, and the entry read (the number of basic lines) are sent to the AI. The physical address are sent to the AI where it is stored in a register. A cache hit/miss signal is

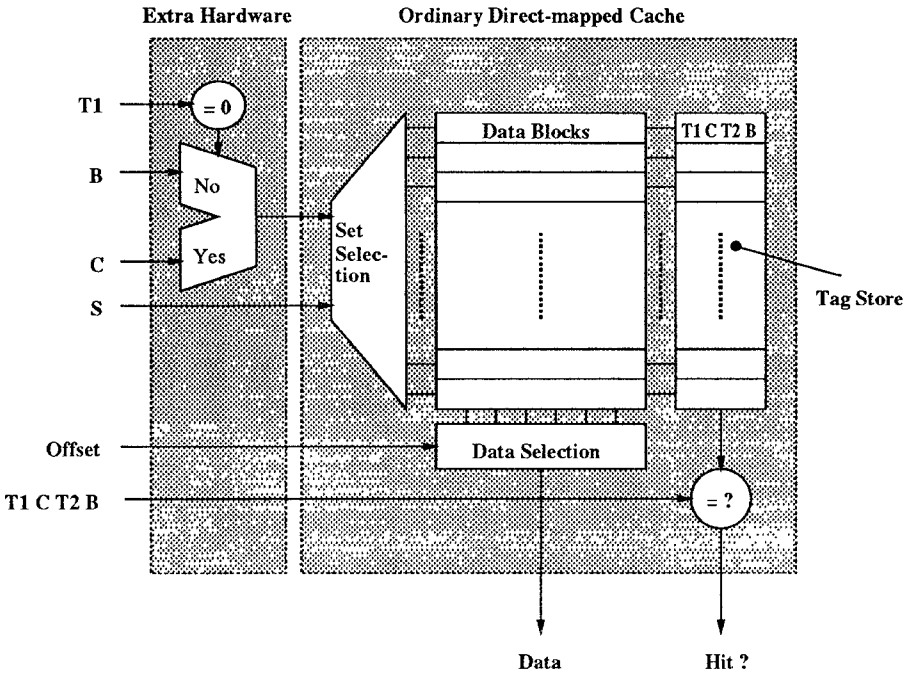

Figure 7: The new cache-configuration for the virtual cache-unit technique.

sent to the replacement controller. In the case of a miss, the replacement controller receives information from the AI about which line that shall be read into the cache, one basic line at a time. The AI begins with the basic line on which the miss occurred, and then continues with the rest (in the case of an entry > 1 from the LST). The processor stall do *not* have to be larger due to a large number of basic lines compared to a single one, because the data accessed may be forwarded directly after the first basic line is arrived, and the rest of the basic lines are transferred in parallel with the processor execution.

If a new cache miss occurs before the line transfer of the previous miss is completed, the basic lines left from the previous miss is preempted, and the basic lines of the new miss is served. This reduces the processor stall due to data cache misses, and the prefetching according to the previous miss seems not that crucial since a completely new data address was accessed by the processor. The average number of processor accesses (instructions + data) between two cache misses can be estimated as $\frac{1}{f_{data}} \times \frac{1}{1-h_{data}}$, where f_{data} is the fraction of accesses that is to data and h_{data} is the hit-ratio in the on-chip data cache. If we have $f_{data} = \frac{1}{3}$ and h_{data} = 0.9 this average is 30 references.

The LST is memory mapped, and the entries can be read or written with normal LOADs and STOREs. Since an entry contains the line size of a certain region of the virtual address space, this region can be changed too. If no entry is found, the default is one basic line. On the other hand is it possible for one entry to point out the whole address space.

4 Methodology

Our descriptions of different methods to increase the hit ratio of a direct-mapped data cache, is supported by simulation results. The simulation environment is described, after which the workload used and measurement results are presented and analyzed.

4.1 Experimental setup

The simulation is based on a program driven simulator [Dah91]. In a program-driven simulator, the processing elements are executing real code, and thereby can run applications, [Fer78]. The simulator used is a RISC-processor simulator with synchronization primitives and executing MC68000 arithmetical operations. The time-slots, i.e. time-granularity, of the simulator is the time between two consecutive memory-references from a processor.

4.2 Workload

Our study concentrates on loops with a large number of iterations, even though some of the methods presented are applicable on other parts of programs too. Typical target applications are numerical algorithms where a large part of the time is spent in numerical loops, and consisting of several array and matrix traversals.

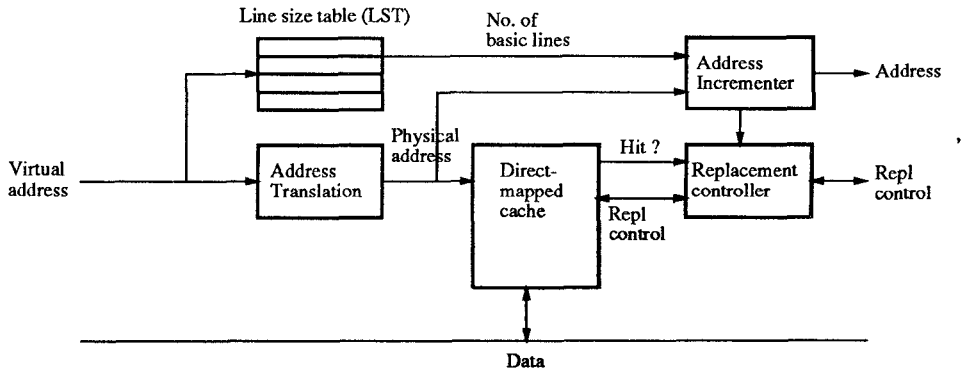

Figure 9: The new cache-control-configuration for the variable line size cache.

One of the most well-established benchmark for numerical loops are the Livermore Loop suite, [McM84]. We chose loops with different characteristics. Several loops were almost the same regarded from the data access point of view, and the loops chosen for analysis became no. 1, 6, 7, and 10. These loops were coded directly in assembly language.

Another suitable application for these methods is a matrix multiplication program, referred to as *matmul*. This is matrix multiplication $C \leftarrow A \times B$, where A, B C are matrices of size 14×14.

Loop 6				
Technique	Configuration: Sets / Line size			
	16/8	64/8	16/16	64/16
Normal	0.837	0.906	0.897	0.946
Cache units	0.866	0.928	0.960	0.962

Table 3: Hit ratio for Livermore loop no. 6, a comparison between a normal direct-mapped cache and the virtual cache unit technique for different cache sizes.

5 Simulation results

5.1 The virtual cache unit technique

The simulation results are shown in the following tables. First we analyzed the data cache hit ratio for the chosen loops at different cache sizes (different line sizes and number of lines), table 2 to 4. The different compiler methods to increase cache performance were used and is compared with the (almost) worst case. The memory allocation of the non-array variables were done in a way that there are no conflicts between their mapping, i.e. there are no bumps between them. For the normal technique, we make a distinction between a "bad" mapping between arrays, i.e. there might exist bumping between arrays even if they have exactly the same indexing pattern, and a "good" mapping between them, i.e. this kind of bumping is avoided if possible. These cases are called worst and best cases for the array mapping without using the virtual cache unit technique. The virtual cache unit technique uses an organization with 2 cache units.

Table 2. Loop 1 contains both non-array variables and arrays. The arrays are indexed in the same way (incremented every new iteration). When using the virtual cache unit technique, the non-array variables are allocated in one cache unit, and the arrays in the other. Now we reach the best results possible with the given line size. These figures show that we reach much better results even with a smaller cache using the new architecture and the virtual cache unit technique for loop 1. The size of the cache is the product of the number of sets and the line size.

Table 3. Loop 6 contains references to two arrays, which are indexed in a non-predictable way. The arrays referenced are *not* traversed in the same way so normally no optimization can be done. By using the cache unit technique, and allocating the arrays in different cache units, we can guarantee that no bumps occur between the two non-predictable arrays. We received a clear increase in the hit-ratio for every cache size simulated.

Loop 7. Loop 7 contains both non-array variables and arrays. The arrays have the same indexing pattern, i.e. it is a trivial task making sure that they are never mapped onto the same set in the cache. If the non-array variables are kept in registers, the virtual cache unit technique doesn't have any effect on the hit-ratio. In this case, it is chosen *not* to use this technique. It is important to notice that it will work exactly the same as a normal direct-mapped cache without any losses in efficiency.

Table 4. Loop 10 contains both non-array variables, arrays, and a matrix. The matrix is referenced as it was different arrays (all the rows are referenced to, with the same column index, in the same iteration). All arrays are indexed in the same way. The number of arrays (including the different

Loop 1							
Technique	Case	Configuration: Sets / Line size					
		8/8	16/8	64/8	8/16	16/16	64/16
Normal	worst	-	0.269	0.282	-	0.286	0.566
Normal	best	-	0.533	0.590	-	0.638	0.923
Cache units		0.892	0.892	0.892	0.946	0.946	0.946

Table 2: Hit ratio for Livermore loop no. 1, a comparison between a normal direct-mapped cache and the virtual cache unit technique for different cache sizes.

Loop 10					
Technique	Case	Configuration: Sets / Line size			
		32/8	64/8	32/16	64/16
Normal	worst	0.434	0.434	0.434	0.504
Normal	best	0.812	0.893	0.945	0.959
Cache units		0.920	0.920	0.969	0.969

Table 4: Hit ratio for Livermore loop no. 10, a comparison between a normal direct-mapped cache and the virtual cache unit technique for different cache sizes.

rows of the matrix) is rather large, and is chosen to be treated as if the number of address registers are not sufficient. This means that some of the array addresses is read in a way that they can be regarded as non-array accesses. When using the cache unit technique, the non-array variables are allocated into one cache unit, and the arrays into the other. With the cache unit technique it is possible to receive an almost optimal hit-ratio (for a given line size) even with rather small caches.

5.2 Performance model for the variable line size cache

In order to better understand the experimental results concerning the variable line size cache, a simple performance model for uniprocessors is presented.

The average access time, $\overline{t_{access}}$, can be calculated according to equation 1. h_{c1} and h_{c2} are hit ratios for the first level and second level caches respectively, t_{c1} is the access time for a cache hit in the first level cache, t_{c2} the time it takes a miss in the first level cache to be handled by the second level cache, and finally $\overline{t_{memory}}$ is the average time it takes the rest of the memory system to handle a miss in the second level cache.

$$\overline{t_{access}} = t_{c1} + (1 - h_{c1}) \times [t_{c2} + (1 - h_{c2}) \times \overline{t_{memory}}] \quad (1)$$

Since we are interested in an on-chip data cache, only the first level cache is of interest. Since we do not want to model different second level caches in order to find out different values on h_{c2}, the environment around the processor chip can be regarded as one memory system with the average time $\overline{t_{miss}}$ to handle a first level cache miss, see equation 2.

$$\overline{t_{access}} = t_{c1} + (1 - h_{c1}) \times \overline{t_{miss}} \quad (2)$$

We define the *slowdown factor* for an access to be the average access time relative the access time for the first level cache, see equation 3.

$$\text{Slowdown} = \frac{\overline{t_{access}}}{t_{c1}} \quad (3)$$

When performing the experiments, we looked at the slowdown factors for different line sizes and different caches sizes when running different programs. When we evaluate the results obtained, we compare the slowdown for a certain line size with the slowdown when having the optimal line size for the same program and cache size. We define the *losses* according to equation 4, which indicate how much slower the program executes just because of a non-optimal line size.

$$\text{Losses}_{linesize} = 1 - \frac{\text{Slowdown}_{linesize}}{\text{Slowdown}_{optimal}} \quad (4)$$

5.3 Results for the variable line size cache

We conducted the experiments with basically three different programs, the Livermore loops 1 and 10, and the matrix multiplication program (matmul). For loop 1, we made a distinction between when the non-array variables were contained in registers (loop1) and when they were not contained in registers (loop1-noreg). Figure 10 shows the slowdown factors obtained from the experiments according to equation 3 above. In these experiments, we had a constant cache size of 256 words, and the cache was direct-mapped. $\overline{t_{miss}}$ is chosen to be 3 in these calculations.

It is obvious that different programs have different optimal line sizes. Significant average performance improvements can be received, if it is able to optimize the line size for a certain program.

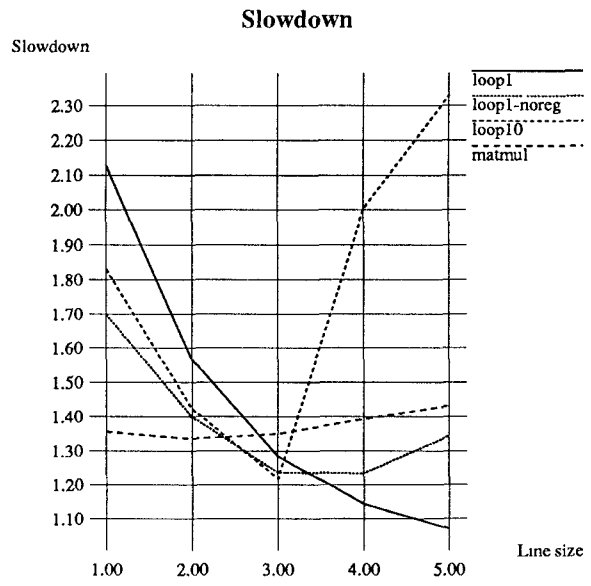

Figure 10: The slowdown for different line sizes and different programs according to equation 3.

Program	Line size				
	2	4	8	16	32
loop 1	98.5%	46.2%	19.9%	6.7%	0
loop 1 (no reg)	37.7%	13.4%	0.2%	0	9.0%
loop 10	50.2%	16.7%	0	64.2%	91.1%
matmul	1.6%	0	1.1%	4.3%	7.2%

Table 5: Losses for different line sizes relative the line size with optimal efficiency according to equation 4.

Table 5 shows the losses in performance compared to the optimal case for different line sizes. A distinct **0** indicates the optimal line size, and thus has no performance loss. Observe that for these examples, the smallest maximum performance loss for the columns is 19.9% (line size 8 words), which is high enough to be taken serious.

6 Discussion and conclusion

We have presented two methods to improve the performance of on-chip data caches. In both methods, it is possible to change the behavior of the cache in a dynamic manner, which is why we call our concept of implementing caches as reconfigurable.

First, we presented a technique to reduce the number of conflict misses in direct-mapped on-chip data caches. The technique called *the virtual cache-unit technique* makes it possible to bound long consecutive parts of the memory onto only a specific part of the direct-mapped cache. This is done by allocating the data to certain virtual addresses. For all data allocated outside the virtual address region of the cache-units, the cache will act exactly as a normal direct-mapped cache, i.e. this technique is optional. Even if the technique seems to offer greater performance improvements for direct-mapped caches, it is usable for set-associative caches as well.

We also proposed an implementation of this technique, where we showed that the additional hardware needed is neglectable. For reasonably large page sizes, the implementation does not degrade the performance, since the additional hardware work in parallel with the address translation.

Simulations showed that for applications that make heavy use of several arrays (i.e. as for Livermore loops), simple memory allocation techniques in conjunction with the virtual cache unit architecture demonstrates significant hit-ratio improvements. We therefore believe that this technique is a viable alternative for achieving a high hit-ratio for direct-mapped caches without introducing associativity. Although software methods could be used to alleviate the same kind of problems, they are limited in the general case.

Second, we presented a variable line (block) size cache. The line size is to be changed explicitly by instructions by writes to a *line size table*, LST, which is memory mapped. The LST contains line sizes for different parts of the virtual memory space. The regions can be chosen, as well as the line size of that region. The smallest line size possible is called a *basic line*, and possible line sizes are multiples of that size. We propose an implementation with no performance losses in speed, compared to a normal architecture with the line size of one basic line. The hardware costs seem small.

We reported simulation results that demonstrated the variety in optimal line sizes for different applications. For the benchmarks we investigated, the best line size choice resulted in at least 19% of losses for one of the applications.

The proposed methods are different techniques for how to implement a data cache architecture, which can increase the performance in a dynamical fashion. They may be used together or separately, but are both examples of reconfigurable cache architectures.

Acknowledgements

The authors are indebted to Anders Ardö for his valuable discussions. This work was supported by the Swedish National Board for Technical Development (STU) under contract number 85-3899 and 87-2427.

References

[BS88] M. Bretemitz and J.P. Shen. Organization of Array Data for Concurrent Memory Access. Technical report, Department of Electrical and Computer Engineering, Carnegie Mellon University, September 1988. Report No. CMUCAD-88-39.

[Dah91] F. Dahlgren. A Program-driven Simulation Model of an MIMD Multiprocessor. In *Proceedings of the 24th Annual Simulation Symposium*, pages 40–49, 1991.

[EK89] S.J. Eggers and R.H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc of ASPLOS-III*, pages 257–270, 1989.

[Fer78] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, 1978.

[FP89] M.K. Farrens and A.R. Pleszkun. Improving Performance of Small On-Chip Instruction Caches. In *Proc of 16'th Int Symposium on Computer Architecture*, pages 234–241, 1989.

[HC89] W.W. Hwu and P.P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proc of 16'th International Symposium on Computer Architecture*, pages 242–251, 1989.

[HP90] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[HS84] M.D. Hill and A.J. Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proc of 11'th International Symposium on Computer Architecture*, pages 158–166, 1984.

[Int90a] Intel. *i486 Microprocessor Hardware Reference Manual*. 1990.

[Int90b] Intel. *i860 Microprocessor Hardware Reference Manual*. 1990.

[McM84] F.H. McMahon. LLNL Fortran Kernels: MFlops. Technical report, Lawrence Livermore Laboratories, March 1984.

[PE88] D.A. Pucknell and K. Eshraghian. *Basic VLSI Design*. Prentice-Hall, 1988.

[SG85] J.E. Smith and J.R. Goodman. Instruction Cache Replacement Policies and Organizations. *IEEE Transactions on Computers*, C-34(3):234–241, March 1985.

[Smi82] A.J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi87] A. J. Smith. Line (Block) Size Choice for CPU Cache Memories. *IEEE Trans on Computers*, C-36(9):1063–1075, 1987.