Hashed Addressed Caches For Embedded Pointer Based Codes

Marian Stanca, Stamatis Vassiliadis, Sorin Cotofana, and Henk Corporaal

Electrical Engineering Department
Delft University of Technology
Delft, The Netherlands
{akela, stamatis, sorin, heco}@cardit.et.tudelft.nl

Abstract. We are proposing a cache addressing scheme based on hashing intended to decrease the miss ratio of small size caches. The main intention is to improve the hit ratio for 'random' patterns pointer memory accesses for embedded (special purpose) system applications. We introduce a hashing scheme, denoted as bit juggling, and measure the effect such a scheme has in the cache access miss ratio. It is shown, for the considered benchmark, that 3-bit bit juggling will reduce the miss ratio for up to 12%, for associative caches of maximum size of 8KBytes when compared to usual cache addressing schemes.

1 Introduction

Embedded systems can be characterized as, at least, mass produced elements of a larger system providing a dedicated, possibly time constrained, service to that system. Additionally, they have a priori known job characteristics, a small time to market window, and, not in the last place, they are very much cost sensitive. Therefore we may apply particular schemes for particular types of applications in order to improve performance. Such schemes are not necessarily delivering improvements in performance for general computing.

The storage and retrieval of items into and from memory problem arises frequently in programming. One possible approach to improve storage and retrieval of information is the hash table method of storage and retrieval. The main characteristic of such schemes is to use the key-value of an item to compute an address for the storage or retrieval of that item. Key values for which the same address is computed are called synonyms. Collisions due to the occurrence of synonymous key values is a major difficulty which can be overcome in various ways. Because the address calculation is generally a randomizing scrambling of the given key value the term of hashing has become the name for this computation. The criterion for categorizing hashing schemes is the way in which collision is resolved[8]. We wish to find a hashing function which maps our given items to addresses such that any given address is not burdened with more than its share of items. The reason for desiring an uniform distribution is to minimize the number of collisions which will occur. Therefore, for all hashing storage and retrieval

schemes a good hashing function is a function which minimizes the number of collisions.

Our interest is in applying hashing functions for cache memory in order to improve the cache hit rate for an application. For every type of hashing function there are programs for which the performance obtained with hashed caches is improved and programs for which the performance with hashed caches is deteriorated. As a consequence of the fact that usual applications are inheriting the characteristics of the Turing Machine approach for computation(using a sequentially accessed tape), they are obeying the spatial and temporal locality law for memory accesses. For applications spawning from different computation paradigms (e.g. Lambda-Calculus[1]), like LISP type of applications, the locality law may not apply any more. While such applications are very rarely encountered, some characteristics of them may appear to real world programs. The main characteristic would be that the next memory access is almost random in relation to the current one. The best substitutes for the purpose of studying such characteristics are the pointer based programs due to memory allocation issues.

In this paper we address the issue of lowering the cache miss ratio for pointer based codes. We assume that we have pointer based applications that are meant to be executed on an architecture that includes a cache memory. Moreover we consider that we are at liberty to intervene on the hardware implementation of that cache memory. More in particular we address the issue of hashing the cache memory for pointer based codes by providing an answer to the following research question:

- Can memory access patterns be changed through hashing so that the cache miss ratio is lowered ?

To evaluate the effectiveness of hashed caches we propose a framework which is based on a simulation tool. Our framework includes a modified cache simulation tool, based on the Simple Scalar[3]. Our experiments suggested the following:

- Hashed caches can lower the cache miss ratio for certain types of applications, while worsening the performance for others.
- The effectiveness of the approach can be assessed only with a priori knowledge of the memory access patterns.
- The implementation cost for the necessary extra hardware is negligeable (mostly inexistent).
- Improvement in miss ratio after applying the hashing scheme is up to 12%.

The current framework is not allowing performance assessments for reconfigurable hashed caches[1] As a future work direction we have identified the study of compiler controlled hashing schemes. Though, in such cases, extra care is necessary as the cache is in the critical path, and reconfigurable hardware has the tendency of being slower than the straight implementation.

[1] Reconfigurable hashed cache is a cache memory that may use different hashing schemes while an application is ran.

The presentation is organized as follows. Section 2 provides some background informations on hashing functions and illustrates the technique used. Section 3 presents the implementation framework, the results and evaluation of the facts. Section 4 consists of conclusions and intended future work.

2 Hashing Functions and Bit Juggling Addressing

Hashing functions are attempting to optimize the process of mapping a big set of items in a much smaller subset without overburdening any item with more than its share. Therefore any function that hashes the cache address into a cache index is optimal for some types of set organization and very far from optimal for others. According to the type of item distribution for which the hashing problem is approached, a classification of functions is possible, e.g.:

- Distribution dependent hashing functions;
- Cluster separating hashing functions;
- Distribution independent hashing functions;
- Index function based hashing functions, et al.

A different type of classification may be based on the type of hashing function (additive, multiplicative, polynomial,...). It is outside the interest of this paper to survey all possible hashing functions, a complete monograph on this subject being[8].

Due to the fact that the hashing function has to be performed in the background, preferably transparent to the user, the implementation of most of the schemes may be very difficult. The reason is the huge amount of items and key values that have to be used. A pragmatic approach intends to observe the statistical phenomenon of which items are more likely to be needed in the cache memory in the mean time, so that successive memory accesses are hits. Therefore our means of obtaining the same type of information will be through experimenting with a simpler technique on particular applications.

The principle of locality indicates that most programs do not access all data or code uniformly. Along with the observation that smaller hardware is expected to be faster suggests that a good attempt to solve the limited storage problem is a hierarchy of memories with different sizes and speeds. The natural approach is for the size to increase and for the speed to decrease as we move further away from the CPU. Cache is generally the name used for the first level of memory hierarchy encountered by an address leaving the computation core. Due to the principle of locality memory management is performed with sets of memory locations named blocks or lines. If a memory access finds a the required data in the cache memory, the process is named a hit, if not is named a miss.

Managing a hierarchy of different memory levels and maximizing the performance requires good handling of a number of issues[6]:

- Where may a line be placed in the upper memory level?
- How may a line be identified as present in the upper memory level?

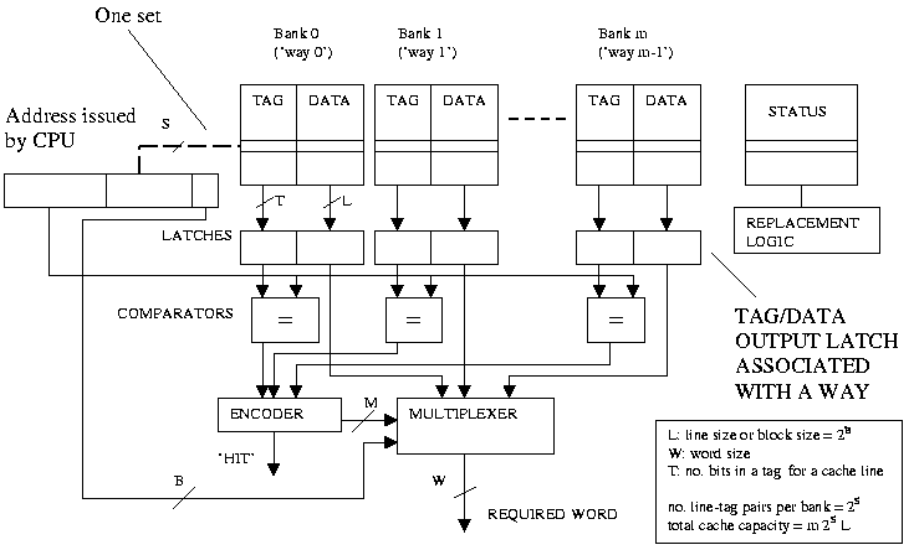

Fig. 1. A m-way set-associative cache organization

- Which line should be replaced on a miss in the upper memory level?
- How are writes handled?
- What are the possible trade-offs to decrease the miss rate?

If a line has only one place where it can be placed in the cache memory the cache is called to be direct mapped, and the mapping is performed with a hashing function as seen in Equation 1.

$$F_{hashing} = \text{(Block address) MOD (\# of blocks in the cache memory)} \quad (1)$$

If a line may be placed in a limited set of places, the cache is said to be set associative, and the mapping is performed with a hashing function as seen in Equation 2.

$$F_{hashing} = \text{(Block address) MOD (\# of sets in the cache memory)} \quad (2)$$

A common organization of a set associative cache memory can be seen in Figure 1. An address issued by the CPU has three different components:

- A tag used to identify if a line is in the cache memory.
- A line index to choose one of m entries from the m-way set associative cache.
- A line offset to choose one memory location from the cache line.

In order to know if the data present in a cache line has valid information, an extra bit, named valid bit, may be added to the tag.

There are a number of possible algorithms regarding the block replacement issue(random, LRU, FIFO, MIN[2]), every one with advantages and disadvantages. Also a write into the memory may be handled in different manners. A

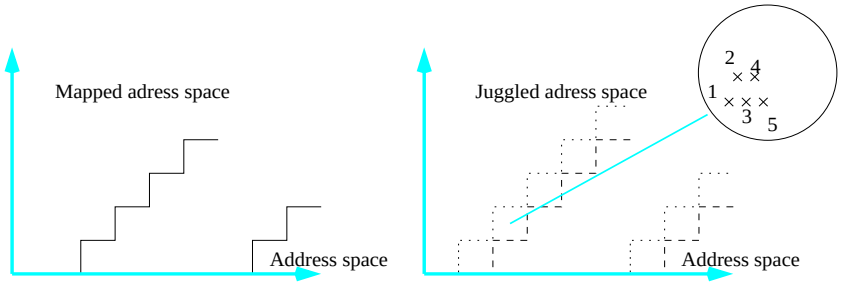

Fig. 2. Normal and hashed address mapping

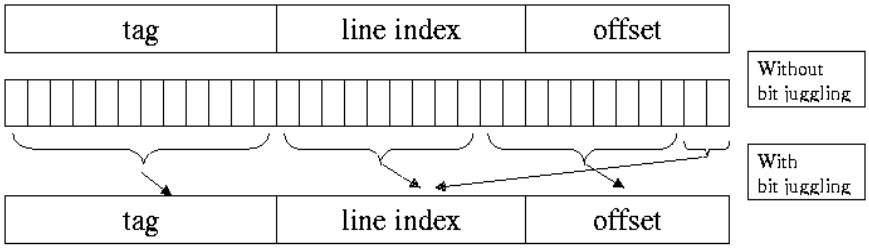

Fig. 3. Tag, line index and offset extracting from issued address

classification may take into account the speed of propagation of that write into the lower levels of the memory (store through, copy back).

Taking advantage of the hierarchy is to minimize traffic between different memory levels. This may be obtained through various techniques, see for example [7, 10, 5, 9].

Our approach is using hashing functions in order to minimize the traffic between two consecutive memory levels. The usual hashing function (MOD) used to map main memory addresses into cache memory addresses can be seen in Figure 2, as well as a different hashing function. Certain bits used for line index or offset are extracted from the address issued by the CPU as seen in Figure 3.

We are introducing a new hashing technique for caches, named bit juggling (BJ). Instead of using the number of necessary bits starting with position 0 to the left for line offset we are using successive bits starting from at least position 1. The necessary tag bits will be the same. The line index bits will be the remaining ones. If, for example, we are using 1-bit BJ, the memory accesses patterns will change as follows: odd address memory accesses in the range of two line size will be stored consecutively in the same line. This holds true for even addresses as well. The line placement mechanism will remain the same except for the notion of successiveness. Successive addresses will now be, in the cache memory,

those separated by $2^{bitjuggled}$ addresses. Accommodating such a technique will necessitate changes to the cache buffers only. In order to simplify the approach we will assume that we may choose the main memory organization's in banks. The number of banks is specified by the number of bits juggled. Therefore, next memory level organization's will allow reading and writing with strides. The line identification process needs the extension of the tag, with a number of redundant bits. Once again the extra number of bits are the number of bits to be juggled. The line replacement mechanism remains the same, according with the cache memory associativity. The improvement in performance due to bit juggling technique on hashed caches are discussed in the next section.

3 Evaluation

In order to evaluate the effectiveness of our approach a simulation tool has been built, by modifying the existing cache simulation tool set Simple Scalar. A set of experiments has been performed using as benchmark the pointer implementation of MPEG2[4]. The Simple Scalar[3] tool set contains a number of simulators that may be used to find profiling informations about programs and their characteristics when ran on various architectures. The tool which interested us most was the cache simulator. We have modified this tool in order to accommodate our technique. The modifications performed were pertaining the addressing mode of the cache, the hit/miss handler routine, and the replacement algorithm.

Issuing and recuperating (internally to the simulation tool) real (in our technique) addresses had to have the bit juggling technique implemented. This was performed by modifying a number of defines from *cache.c*. Because of the fact that successive addresses for the program were not kept in the cache memory successively, the hit or miss decision had to be modified according to the number of bits to be juggled. In the original tool, the decision of hit was taken based on a range checker. If an access was within line size from the currently accessed cache line a decision of fast hit was taken. If an access was within line size from an existing in the cache memory cache line a decision of hit was taken. If none of the above decisions was taken then a decision of cache miss was taken. Due to the organization of the existing code, we accommodated the same techniques for hit decision. Hit decision for a memory access was performed in the same manner, except for the notion of belonging to the same cache line. 'Successiveness' in the same cache line in the original version was replaced with 'successiveness' according to the bit juggling notion. A similar change to the issuing and recuperating addresses mechanism was implemented. This approach allowed to accommodate in our technique a consistent replacement algorithm. The bit juggling technique for hashed caches places successively in the same cache line memory content separated by $2^{bitjuggled}$ addresses. It has to be noted that performance assessments on the simulation tool are translated to the original tool set.

Choosing benchmarks to test our technique was a delicate issue. 'Pointer based' and 'big enough to observe relevant statistical phenomenons' were two key characteristics. Translating the C implementation pointer based of MPEG2[4]

Table 1. Data cache miss rates for various configurations

Line size →	8	8	16	16	32	32
Cache size	-	BJ3	-	BJ3	-	BJ3
512Bytes(2)	**45.10**	**36.00**	**43.65**	**35.26**	**41.43**	**34.82**
4KBytes(2)	**39.11**	**28.76**	**37.15**	**27.93**	**36.72**	**25.12**
8KBytes(2)	**34.38**	**24.77**	**33.87**	**22.15**	**33.15**	**20.10**
512Bytes(4)	**43.44**	**33.17**	**40.34**	**32.23**	**38.52**	**29.48**
4KBytes(4)	**37.12**	**27.19**	**35.40**	**26.20**	**31.21**	**23.17**
8KBytes(4)	**29.29**	**18.31**	**26.14**	**16.26**	**22.46**	**12.32**

into MIPS IV assembler needed by the cache simulation program must be perform with a re-targeted gcc compiler, provided with Simple Scalar. This translated version has around 13 Ginstructions and 4.5 Gdata accesses for a small mpeg stream of frames. These figures made the MPEG2 pointer implementation an ideal candidate to test our technique. After obtaining an instrumented executable, namely the MPEG2 encoder / decoder, we simulated it with our modified tool. The results presented are for the MPEG2 decoder, the encoder shows similar results and performance improvements patterns. Along with the cache simulator a computation architecture simulator is ran in order to obtain the addresses required by the cache simulator. No modifications have been performed to the computation architecture simulator as the modifications were regarding only the cache. This simulator provides output for the instrumented executable, if the original program to be simulated was intended to have output. It has to be noted that running a normally compiled program and the simulator with an instrumented executable with the same input provides the same output. We simulated an MPEG2 decoder with a 476 frames mpeg stream, with 300 x 200 pixels image size. The results are presented in Table 1 for a number of cache sizes, cache associativities of 2 and 4, and for various cache line sizes. Due to implementation particularities cache miss ratio varies proportionally with the frame stream size(weak) and frame size(strong). The data stream is organized in 8 x 8Bytes objects(searched and identified in successive frames), or smaller. Due to this reason we had to keep the line size smaller than the customary 32-128Bytes in general computing in order to observe relevant patterns in memory accesses in various parts of the application. Because of the small cache line size the number of first reference cache misses was increased. Also, the simulations were performed without a second level cache memory, so the performance of the memory hierarchy is still far from the expected 5-10% miss rate. It has to be noted that the pattern in performance figures for this application will be maintained for bigger cache line sizes, in relative terms. Due to the data stream organization the 3-bits BJ was our prime candidate to test our technique and the improvement in performance was up to 12%.

More results will be reported in the final version of the paper.

Table 2. Data cache miss rates for 64Bytes cache size, 8Bytes line size

	4 entries associativity 2	2 entries associativity 4
No Bit Juggling	34.91	32.34
Bit Juggling 3bits	25.17	21.75

For the direct mapped cache, a cache line can be stored in one place only and collisions are solved by replacing the old cache line with the new one. Because of this fact a change in the memory access patterns for caches with associativity of 1 will not show notable modifications in performance. Therefore we concentrate our efforts on cache memories with greater associativity.

Caches smaller then 4KBytes are not likely to be found in general purpose computing but can be found in embedded applications. In our simulations we assumed small cache sizes intended for application specific inexpensive engines. For comparison reasons we have included some simulation figures for a very small cache (64Bytes) in Table 2, for a line size of 8Bytes and a mpeg stream containing only 34 frames, with 150 x 100 pixels image size.

We could have artificially create a bigger/smaller pointer application or increase/decrease the number of frames in the mpeg stream, or the frame size, but we feel confident that the established pattern in performance readings for this application remains the same. We can conclude, after studying the results, that the MPEG2 pointer based implementation should be ran on an architecture with a hashed cache with 3 bits bit juggling technique.

The same process may be applied for any application. The bit juggling technique has the same characteristics as hashing functions. It may offer gains or degradations in performance depending on the application. Therefore the possible improvement or degradation in performance when running an application on an architecture with a hashed cache may vary with the application. Applying optimally this technique assumes a priori simulation. The results are compared and the best option is chosen. The characteristic of applications prone to performance improvement through this technique is that they do not obey the locality law, in the classical nearest neighbor sense. One possibility is to have successive random memory accesses. Another is to have array based or record based codes that are accessed with strides. In this case deterministic determination of the number of bits to be juggled is possible simply from inspection.

4 Conclusions and Future Work

A new technique to decrease the cache miss ratio, denoted bit juggling, for pointer based codes has been introduced. Bit juggling has the characteristic of any hashing process, namely is offering improvement or worsening of performance according to the application. For a sample of pointer based code it has been shown that three bits bit juggling is offering good relative performance

improvements. Using the same technique with lesser bits shows degradation in performance. The success in applicability of this technique is related with the amount of information on memory access patterns a priori known. Embedded applications are, therefore, a primary target.

The most that may be obtained from such a technique is a reconfigurable scheme, with compiler control. This offers the advantage of being able to estimate and modify the hashing scheme at run time. The main disadvantage is in the fact that the cache memory has to be flushed when modifying the hashing function. While taking advantage of the a priori knowledge about memory accesses may offer the possibility to trigger the cache flushing in advance (in order to reduce the penalty for this operation), reconfigurable still affects the cycle time. Reconfiguration for recuperation of line offset bits may be obtained in logarithmic depth, though, due to the fact that the cache may be in the critical path, deterioration of cycle time is expected. No simulation may be performed for a reconfigurable scheme in the current framework.

Hashing function is a very vast research domain, covered by various materials, therefore the domain of hashed caches is equally vast. As performance improvement is depending on the application, sampling the process of choosing a correct hashed cache memory for an application is a sufficient answer to the research question.

A promising research approach, for hashed caches, is having an adaptive hashing function, obtained through observing recent memory accesses behavior . This will be generally applicable, and will observe and speculate spatial and temporal locality in memory accesses much better. An on-line changing of the hashing function, may be an exclusive or between the extended tags (as defined in section 2) for successive memory accesses. This type of hashed caches are much more difficult to simulate. The expected effect is obtaining at least the same improvements in performance as buffered cache schemes [11].

References

1. H. P. Barendregt. Functional programming and lambda calculus. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 7, pages 321–363. The MIT Press, New York, NY, 1990.
2. L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
3. D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
4. M. S. S. Group. Mpeg-2 video codec, http://www.mpeg.org/index.html/mssg/source, 1996.
5. M. Gupta and D. A. Padua. Effects of program parallelization and stripmining transformation on cache performance in a multiprocessor. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, Architecture, pages I–301–I–304, Boca Raton, FL, Aug. 1991. CRC Press.

6. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

7. D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 252–263, June 2–4 1997.

8. G. D. Knott. Hashing functions. *The Computer Journal*, 18(3):265–278, Aug. 1975.

9. L. I. Kontothanassis, M. L. Scott, R. A. Sugumar, G. J. Faanes, and J. E. Smith. Cache performance in vector supercomputers. In *Supercomputing '94*. IEEE Computer Society, 1994.

10. P. Krishnan and J. S. Vitter. Optimal prediction for prefetching in the worst case. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 392–401, 23–25 Jan. 1994.

11. V. Srinivasan. Improving performance of an L1 cache with an associated buffer. Technical Report CSE-TR-361-98, University of Michigan Department of Electrical Engineering and Computer Science, Mar. 20, 1998.