Data cache performance evaluation: the impact of architectural issues

Ahmed Patel, Rinat Khoussainov, Nikita Schmidt
Computer Networks and Distributed Systems Research Group
Department of Computer Science, University College Dublin
Belfield, Dublin 4, Ireland

ABSTRACT

This paper considers a case study of cache usage showing the importance of some architecturally specific issues for performance improvement on the basis of quantitative analysis. Existing cache performance analysis methods are discussed together with the design peculiarities of the hardware used. Guidelines are given to software developers on the application of traditional cache usage techniques with respect to particular hardware designs. The importance of quantitative analysis of cache performance is outlined using a real system example.

Keywords: Performance, Memory Access, Cache, RISC, Quantitative Analysis

1. INTRODUCTION

The explosive growth of computational power is driven by ever increasing industry demands made by advance applications. Having started from simple tasks such as matrix operations or statistical calculations, computer systems are now used to solve problems as complex as 3D image rendering or weather forecasting. The performance requirements for modern computer systems continue to grow infinitely.

Unfortunately, the epoch when the performance of a computer system was determined mainly by the signal distribution delays in logical circuits of the system is over. Technical achievements of semiconductor industry all over the world have drastically reduced these delays, allowing for ever increasing clock frequencies used in modern computer systems. For instance, the first single chip CPU, Intel 4004, introduced in 1971, ran at clock rate of 740 kHz [1]. Nowadays CPUs can run at speeds as high as 600 MHz [2]. One-gigahertz microprocessors have already been announced by DIGITAL [3] and IBM. Thus, the performance of logical circuitry has increased up to 1000 times over the last three decades.

However, the performance increase of other architectural components of computer systems has been significantly behind the CPU improvements. Nowadays the key limitation is imposed by the memory subsystem [4]. Processors can perform operations much faster than the access time of large-capacity memory. Although semiconductor memory, which can operate at speeds comparable with the operation of the processor, exists, it is too expensive to be used as the main high volume memory for general-purpose computer systems. Besides, it would not be technologically justifiable due to the physical constraints involved.

The industry's proposed solution for the problem described above is the utilisation of cache memory, high-performance small-capacity storage serving as an intermediary between the CPU and the main memory. A cache was first used in a commercial computer system by IBM in the IBM System/360 Model 85 [4]. Cache system records the results of a number of recent memory references, allowing for fast access to the data that have recently been read or written. This may lead to a significant increase in the overall performance.

However, the performance improvements gained from the use of cache memory are strongly affected by a number of factors:

- organisation of the cache subsystem and the caching algorithms;
- overall system design and the place of the cache memory in it;
- structure and design of the software running on the system.

Thus, the overall system performance is governed not only by the clock frequencies of CPU and caches, but also by the above listed factors. To achieve maximum effect, a programmer has to take into account cache related issues when designing, developing and tuning the software. In order to produce efficient code it is important to be able to manage and predict the influence of different cache control settings and programming techniques on the cache system behaviour [5].

This paper considers a case study of cache usage showing the importance of the above mentioned issues for performance improvement on the basis of quantitative analysis. Existing cache performance analysis methods are discussed together with the design peculiarities of the hardware used. Guidelines are given to software developers on the application of traditional cache usage techniques with respect to particular hardware designs.

2. SURVEY OF CACHE ANALYSIS TECHNIQUES

Existing methods of cache performance analysis are subdivided into the following groups [6]:

- analysis by measurement;
- analysis by synthesis;
- analysis by simulation;
- analysis by graph techniques.

Analysis by measurement [7] is performed on the actual system. This approach is easily applicable and allows to analyse complex algorithms such as recursion, unbounded loops, which is usually difficult or even impossible to analyse using other techniques. A major disadvantage of this method is that the values being measured are affected by the process itself. This, however, can be reduced by using hardware tools like performance counters [8], which can often be found in modern computer architectures.

Analysis by synthesis [9] is a hardware-assisted approach to improve the performance by prefetching instruction blocks into the instruction cache. Improvements are achieved by giving the software applications more control over the cache subsystem. However, a specialised hardware is required.

Analysis by simulation calculates the number of cache hits and misses by executing program on a cache simulator. It gives the precise numbers of hits and misses, which can either be used to fine tune the software or to predict execution time. It can be easily tuned to different cache parameters, which makes it particularly useful for performance evaluation of different computer architectures, including experimental boards not yet implemented in silicon.

Graph based analysis [6] is a new promising technique which attempts to analyse the algorithm in order to determine cache requirements and estimated behaviour. On the basis of program data and control flow and the basic block detection, cache line allocation is predicted so the resulting execution time can be analysed. This algorithm can also perform variable grouping for optimal cache line allocation, similar the register allocation algorithms used by compilers.

Two different cache implementation approaches are used in computer systems [4]:

- unified instruction and data cache;
- separate instruction and data caches.

These schemes are sometimes referred to as *von Neumann* and *Harvard* schemes respectively, by analogy to the two basic computer designs. Nowadays, Harvard caches are the most commonly used due to the following possible advantages:

- data and instructions which would have shared the same cache line in von Neumann model do not collide in Harvard caches;
- they are much easier to predict and analyse than von Neumann caches.

At the same time, Harvard model has some slight leaks, which are however not critical:

- caches may not achieve 100% utilisation: instruction caches can not take load from data caches and vice versa;
- difficulties in writing self-modifying code and the necessity to take special precautions when loading executable images.

The techniques described above are mostly concerned with instruction cache utilisation for the Harvard scheme. They (except for the first one, analysis by measurement) are not easily applicable to complex pieces of software: e.g., those consisting of numerous small basic blocks and having hardly predictable control flow traces, or applications that can only be run in real time. However, they often serve as a source of ideas and guidelines for use in common applications to determine the most efficient cache parameters and settings for a particular system.

The rest of the paper demonstrates how analysis by measurement has been used to investigate efficiency of traditionally employed guidelines for data cache settings using a Motorola MPC860 based system as an example.

3. SYSTEM DESCRIPTION

The MPC860 PowerPC™ Quad Integrated Communication Controller (PowerQUICC) is a versatile one-chip integrated microprocessor and peripheral combination that can be used in a variety of controller application. In particular, excels in both communications and networking systems [10]. The CPU on the MPC860 is a 32-bit PowerPC implementation that incorporates memory management units (MMUs) and instruction and data caches. The following is a list of the MPC860's features that we find important in our work:

- 32-bit address and data buses;
- flexible memory management:
- 32-entry fully associative instruction translation lookaside buffers (TLBs);
- 32-entry fully associative data translation lookaside buffers;
- attribute support for trapping, write-through, cache inhibit and memory-mapped I/O;
- 4k, 16k, 512k or 8M page size support;
- 4-kbyte physical address, two-way, set associative data cache;
- 4-kbyte physical address, two-way, set associative instruction cache.

The MPC860 data cache is a 4-kbyte two-way set associative cache. The cache organisation is 128 sets, 2 lines per set and 4 words (16 bytes) per line. Cache lines are aligned on 16-byte boundaries in memory. Two state bits are included in each cache line, implementing invalid, modified-valid and unmodified-valid states of the data cache line. The cache coherency in multiprocessor environment is maintained by the software and supported by a fast hardware invalidation capability. The cache is designed for both write-back and write-through operation and a least recently used (LRU) replacement algorithm is used (see Figure 1).

The MPC860 implements a virtual memory management scheme that provides cache control and effective-to-physical address translation. MMU cache control allows setting of cache policy for each page independently.

According to the MPC860 specification, it is designed to be used primarily as a core of data communication systems. Hence the following data cache performance optimisation guidelines are recommended for software development for this platform [11]:

- Write-back policy should be used for all "software only" data.

  "Software only" means data used solely by the CPU: typically stack, software scratch variables, data tables and any other types of data that will never be passed to output.

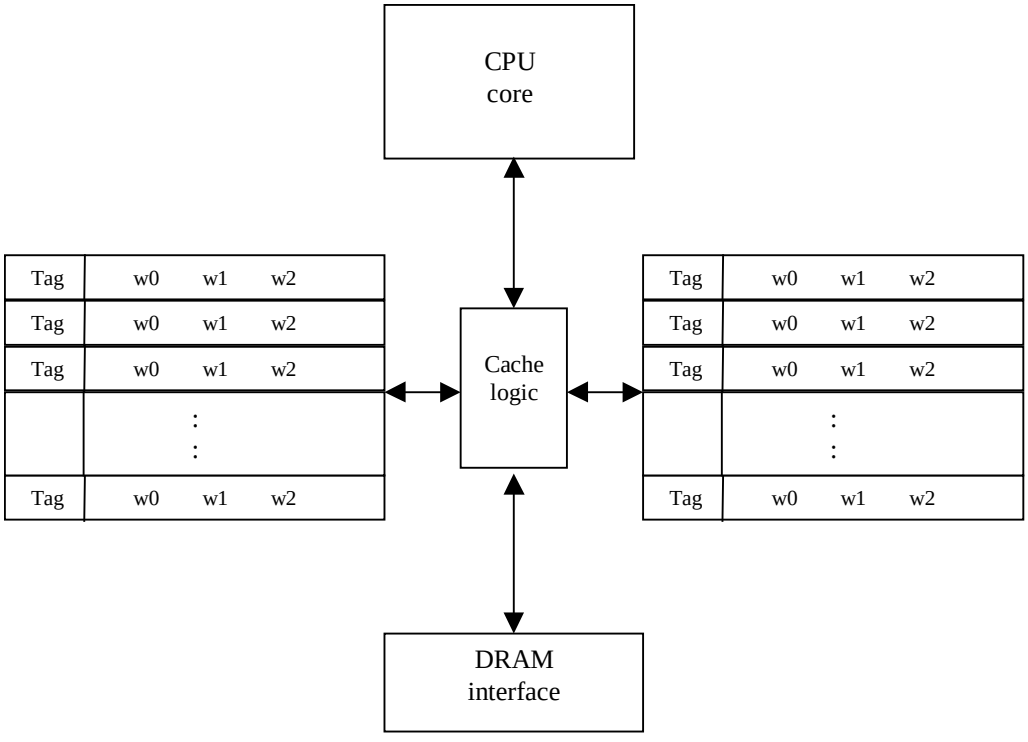

Figure 1: MPC860 cache structure

- Write-through policy should be used for transmit data, i.e. data destined for external devices, such as communication controllers or DMA. As the external devices have no access to the CPU data cache, it is important to maintain strict coherency between memory and data cache.
- Data cache should be inhibited for receive data, that is, data coming from external devices. This way cache contents will not be replaced by data which are not supposed to be used frequently.
- Data cache should be inhibited for data that are known to be used only once.

The next sections of this paper give an experimental analysis of these recommendations on the basis of performance evaluation.

4. BENCHMARKING SUITE

Our benchmarking suite includes:

- MMU set-up code that establishes cache policy configuration for the chosen benchmarking mode.
- Time stamping routines.
- A sample piece of code whose execution time is used as the benchmarking result.

This piece of code consists of a simple loop, which reads in an array of "sample data" (see Figure 2).

```
extern unsigned char sample[];
extern int sample_len;

main ()
{
       int i, k;

       MMU_setup ();

       k = 0;

       start_timer ();

       for (i=0; i<sample_len; i++) {
              k += sample[i];
       }

       stop_timer (sample_len);

       return 0;
}
```

Figure 2: The benchmark source code

Several tests were performed, which differ in the cache policies assigned to the "sample data" memory region. The results of the tests are shown in Table 1 below.

Table 1: Comparison of the performance for different cache policies

Cache policy for the sample data	Cache inhibit	Write-through	Write-back
Process speed	407088 bytes/sec	534836 bytes/sec	534624 bytes/sec
Relative performance	100%	131%	131%

The results obtained clearly demonstrate that the recommended guidelines achieve suboptimal performance for this particular case. Indeed, the sample data are used only once during the lifetime of the benchmarking loop, and yet a significant performance increase is achieved when the data cache is enabled.

This may look unexpectedly at the first glance, although the explanation can be seen from the memory access pattern. For each byte requested by the program, one word (4 bytes) is to be fetched from the memory. When the data cache is enabled, a cached copy of the word is used to obtain the three subsequent bytes, provided that the bytes are requested in order. This significantly reduces the number of memory accesses when the data access pattern is contiguous and the access unit is less than a word. Besides, as the data is fetched into the cache by 16-byte lines, further reduction of the delay imposed by the memory access is possible.

5. BENCHMARKING OF A REAL APPLICATION CODE

The influence of the above mentioned effect is now analysed for V.42bis compression and decompression code [12]. This code is a good example of a typical data communication application characterised by a nearly contiguous memory access pattern with small units.

The code benchmarked is an actual V.42bis compressor, fetching source data from the system memory and

feeding the compressed bytestream to a V.42bis decompressor. The decompressor's output is discarded.

Two different memory regions are distinguished: the "sample data" region, containing the source for the compressor, and the "internal data" region, comprising of the compressor's and decompressor's local variables and data structures such as codeword trees. This is illustrated in Figure 3.

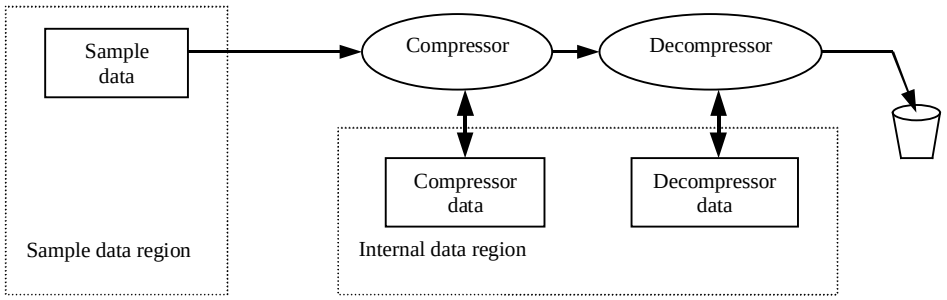

Figure 3: V.42bis code measurement scheme

The measurement results are presented in Table 2.

Table 2: V.42bis code performance (bytes/sec)

Sample \ Internal Data	Cache inhibit	Write-through	Write-back
Cache inhibit	20656	39328	52848
Write-through	20932	40316	54652
Write-back	20932	40316	54652

This table clearly shows that the considerations noticed above for a simple benchmarking example are still applicable to much less straightforward algorithms. The benefits of caching in the case of a contiguous memory access pattern often prevail over the theoretical advantage of keeping the cache clean of data that is "accessed only once".

6. CONCLUSION

In this paper we have discussed general cache performance improvement techniques and their practical applicability. Several cache performance analysis methods were assessed. It was shown that when optimising cache usage by a software application, many issues have to be taken into account. In particular, the utilisation of cache-friendly memory access patterns was demonstrated to prevail over the general advice to keep the cache clean of infrequently used data.

Quantitative analysis was employed in order to obtain accurate results, which were highly dependent on the architectural peculiarities of a real system. This evaluation approach allowed correcting cache management recommendations based on theoretical speculations.

Future work is continuing towards modelling of the achieved results to refine the cache behaviour models used for performance tuning and prediction.

From the research conducted authors believe that similar quantitative based approaches can be used in other case studies and increase the accuracy of the performance estimation models in use.

7. REFERENCES

[1] *Great Microprocessors of the Past and Present.* http://infopad.eecs.berkeley.edu/CIC/archive/cpu_history.html

[2] *Alpha 21164 Microprocessor.* Alpha Microprocessor Products, DIGITAL Semiconductor, 1997. http://www.digital.com/semiconductor/alpha/dsc-21164.html

[3] *DIGITAL to break 1,000 MHz barrier with high-powered new generation of Alpha architecture.* DIGITAL Semiconductor, 1998. http://www.digital.com/semiconductor/news/press02feb.htm

[4] Barry Wilkinson. *Computer architecture: design and performance*. Prentice Hall Europe, 1996.

[5] David Mosberger-Tang. *Linux/Alpha, or How to make your applications fly*. LinuxExpo' 97 proceedings, 1997.

[6] Swagato Basumallick, Kelvin Nilsen. *Cache Issues in Real-Time Systems.* Iowa State University, May'94.

[7] Kenny, K. B. & Lin, K. *A measurement-based performance analyser for real-time programs.* UIUC Report No.: UIUCCDCS-R-90-1606.

[8] *Alpha Architecture Handbook.* Digital Equipment Corporation, 1996.

[9] Lee, M., Min, S. L., Park, C. Y., Bae, Y. H., Shin, H., & Kim, C. S. *A Dual-mode Instruction Prefetch Scheme for Improved Worst Case and Average Case Program Execution Times.* Proc. Real-Time Systems Symposium, 1993.

[10] *MPC860 PowerQUICC User's Manual*. Motorola, 1996.

[11] *MPC860 Performance Checklist*. Revision 2, Motorola, 1997. http://www.mot.com/netcomm

[12] *Data compression procedures for data circuit terminating equipment (DCE) using error correction procedures*. Recommendation V.42bis. CCITT, Geneva, 1990.