Tackling Cache-Line Stealing Effects Using Run-Time Adaptation

Stéphane Zuckerman and William Jalby

University of Versailles Saint-Quentin-en-Yvelines, France
{stephane.zuckerman,william.jalby}@prism.uvsq.fr

Abstract. Modern multicore processors are now found in mainstream systems, as well as supercomputers. They usually embed prefetching facilities to hide memory stalls. While very useful in general, there are some cases where such mechanisms can actually hamper performance, as is the case with cache-line stealing. This paper characterizes and quantifies cache-line stealing, and shows it can induce huge slowdowns – down to almost 65%. Several solutions are examined, ranging from deactivation of hardware prefetching to array reshaping. Such solutions bring between 10% and 65% speedups in the best cases. In order to apply these transformations where they are relevant, we use run-time measurements and adaptive methods to generate code wrappers to be used only when prefetching hurts performance.

1 Introduction

Multicore processors are from now on the norm for almost any kind of computer-based system. They are used in high-performance computing, as well as domestic usage or even embedded systems. The increasing gap between how fast a CPU is and how fast data can be accessed from memory becomes even more problematic in this context: the old hardware and software techniques used to hide it must prove themselves useful in the wake of some kind of "multicore/manycore revolution". Indeed while the well-known gap between a unicore processor and memory is more or less solved thanks to well-known latency-hiding techniques (either in hardware or in software), the multiplication of the number of cores per chip tend to lengthen a given core's memory latencies, while reducing its effective bandwidth.

From the hardware side, the use of memory caches does a lot to make the memory wall [27] and the gap it causes become smaller and narrower. It effectively hides the latency caused by memory accesses. However, when multiprocessor or multicore systems are involved, it also means ensuring data coherence between the various (separate) caches. Hence the apparition of cache coherence protocols such as MSI, MESI, MOESI, MESIF, etc. [15]. With the advent of multiprocessor (and now multicore) systems a new kind of problem occurred: false-sharing, i.e. the fact that two processors (or cores) write to different values which are contained in the same cache-line. The negative impact of false-sharing on performance for multiprocessor and multicore systems has been extensively studied [9],

K. Cooper, J. Mellor-Crummey, and V. Sarkar (Eds.): LCPC 2010, LNCS 6548, pp. 62–76, 2011.

and multiple techniques have been devised to detect (via memory tracing for example), or even avoid false-sharing altogether (through data structure reshaping, loop transformations, etc.).

Another well-known technique which helped reduce the gap between memory and CPU is data prefetching. In a single-core context, data prefetching (whether performed automatically in hardware, or inserted by the compiler or the user in software) has shown to be a formidable ally to hide latencies, even though it induces a higher bandwidth usage in the case of hardware prefetching [8] or instruction overhead for software prefetching. By prefetching data into caches ahead of time, memory can be accessed much faster, thanks to well placed prefetch orders. However once again, the advent of multiprocessor and multicore systems tends to introduce difficulties. Among them is *Cache-Line Stealing*(CLS), which we present in Section 2.

To our great surprise, we could not find any paper detailing how too aggressive a data prefetching policy could indeed slow down an application by "stealing" another thread's cache-line. The closest to our definition can be found in a paper by Hyde et al [9]. CLS is not so frequent that deactivating prefetch mechanisms altogether to solve cache-line stealing is considered an option, as hardware or software prefetching have many legitimate uses in the life of a running program. Hence it is necessary to remove prefetching only in those portions of code where it is known that no prefetching will make performance go up.

This paper's contributions are twofold:

- It characterizes and quantifies cache-line stealing.
- It proposes to solve CLS through adaptive compilation methods, deactivating prefetching only when CLS actually hurts performance.

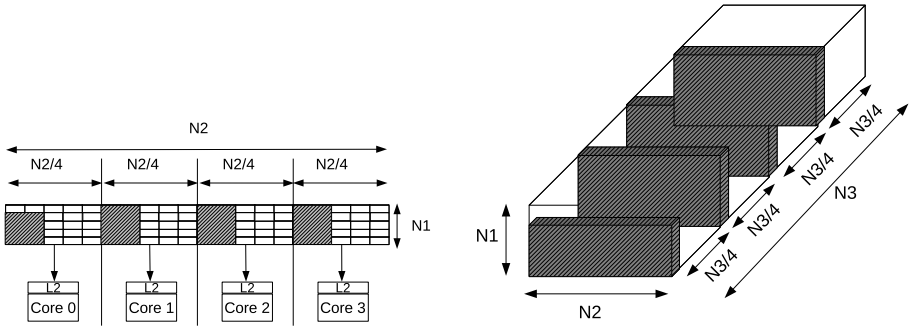
The remainder of this paper is as follows: Section 2 exposes an example that motivates this research and describes what cache-line stealing is, and it occurs, as well as experimental data; Section 3 presents several leads to solve cache-line stealing problems; Section 4 describes the use of adaptive methods to generate prefetch-less kernels when needed; Section 5 presents work related to this article; finally Section 6 presents our conclusions.

2 Motivation

2.1 What Cache-Line Stealing Is, and When It Occurs

Cache-line stealing (CLS) happens when a given core asks for a cache-line in advance, retrieves it and copies its data into its cache hierarchy while it does not need the data contained in this cache-line. At the same time, another core does need this cache-line and has already retrieved it or is about to. By prefetching an unnecessary cache-line, additional memory and coherency message traffic are incurred.

CLS happens only when certain conditions are met. First, accessing data in the "right" storage way usually does not provoke a significant cache-line stealing



(a) 2D array: "overprefetched" surface (b) 3D array: "overprefetched" volume

Fig. 1. Cache-line stealing: general idea. CLS on a 2D array (left) and a 3D one (right). Dark grayed areas are the ones which are prefetched by an adjacent core but are not needed by it. As the array is supposed to be one whole contiguous block, there is a wrap around where the first block of data (on the far left) is partially prefetched by a sibling thread, just like any other block.

event. CLS mostly happens when a multidimensional array is accessed in the "wrong" dimension. For example, accessing a 2D array in C through its columns could trigger CLS in a multithreaded environment. Hence partitioning the input data set along the columns of a 2D array in C for each thread to process could lead to additional memory traffic.

Figure 1 shows an example of what would occur in a 2D array as well as in a 3D one. Each thread prefetches data belonging to one of its siblings, copying said data into its cache hierarchy. This then triggers cache coherence messages, where cache-lines will be falsely tagged as shared, invalid, etc.. How negative an impact that kind of extra-traffic (in terms of data movement and coherence messages) will have on performance depends on the prefetching policy: if the hardware prefetcher systematically goes to prefetch the same stream, then this phenomenon will be amplified – and the more cores are used in the program, the more likely extra-traffic will occur.

Knowing how many extra cache-lines ($ExtraCL$) will be loaded in addition to everything else in an n-dimensional array sums up to solving

$$ExtraCL = NbOfStreams * \prod_{i=2}^{n} N_i * dist_{prefetch}(N_1 - 1)$$

n is the number of dimensions of the array, and N_i are the different dimensions of said array. N_n is the "innermost" dimension, i.e. if the array was traversed in the correct way (along the cache-lines), this dimension would be traversed in the innermost loop of the loop nest. $dist_{prefetch}$ is the prefetch distance. $NbOfStreams$ is the number of different memory streams which are concurrently being accessed. In a hardware-based prefetching mechanism, a limited amount of streams can be accessed (typically, from 2 to 8).

Computing the amount of data stolen can prove difficult: in a hardware-based prefetch mechanism, the prefetch distance is not necessarily given by the vendor. In a software context, it is much easier to do, provided the arrays are accessed directly.

2.2 Experimental Setup

Our experimental setup consists of a Xeon (2.0 Ghz) Core 2 system with 4 cores in total (2×2 cores), composed of two dual-core processors sharing 2 GB of main memory. The L1D caches(32 KB) are private to every core while the L2 caches (unified, holding both instructions and data) are shared by groups of 2 cores (called *adjacent cores*), their size being 4 MB. Core 2 cores are out-of-order, superscalar processors, embedding SIMD instructions (SSE). The two levels of caches embed hardware prefetchers capable of analyzing memory address streams and triggering prefetches called DPL. They also embed a prefetching mechanism called "adjacent cache-line fetch", which systematically fetches a cache-line situated right after the one being fetched [8].

Intel's C compiler (`icc` v11.1) is the reference compiler used in our experiments. Our main way of measurement was the `RDTSC` instruction, which gives an exact count of elapsed cycles. We also used Intel's Performance Tuning Utility (PTU) to measure cycle and instruction counts. Finally, `icc`'s OpenMP features were used to make the parallelizations used in the experiments.

With respect to experimental methodology, each kernel was run once (to put data into the cache hierarchy), then 2000 times and averaged (with respect to the cycles count), and each run is repeated 100 times. Among these runs, the lowest averaged result (in cycles) is then selected – i.e. the best performance is kept. This ensures that noise (due to the operating system for example) is kept minimal during the experiments. Moreover, the repetition loop has an interesting side effect on our study: if the arrays are small enough to fit in a given cache level (for example L2), the first iteration will initially fetch the arrays into the L2 cache, but the following iterations will already find them in L2. All the arrays are aligned on a page boundary basis. The fact that there are NUMA accesses is of no concern here, as data fit into caches, and there is sufficient repetitions performed.

Finally, the dimensions of the arrays have been chosen so that there is no inherent false-sharing when dividing work among threads.

2.3 Experimental Analysis of Cache-Line Stealing

To show evidence of cache-line stealing, two very simple kernels were used. As the ratio between loads and stores has an impact on performance, we started with a simple "store only" kernel (`memset2D`), then a "load-store" one (`memcpy2D`). Finally, we used a "one store, many loads" kernel, where the amount of loads varies to observe its impact on CLS. For the latter case, stencil operator kernels were used.

Table 1. Impact of hardware prefetching on performance for `memset2D` (left table) and `memcpy2D` (right table), with prefetching either turned ON or OFF. In each table columns labeled ON (resp. OFF) display the average performance in cycles per store for `memset2D` and per load-store for `memcpy2D`: lower is better. The last column labeled "Speedup" shows the impact of turning off prefetch: a speedup value greater than 1 shows that *turning off prefetch is beneficial*. i refers to the number of threads at runtime. For each kernel, the same amount of work was given to each thread to perform. In the case of `memset2D` and `memcpy2D`, a $k \times N$ subarray (or subarrays) was fed to each thread, $8 \leq k \leq 30$ and $800 \leq N \leq 3200$ per thread.

(a) Impact of hardware prefetching for `memset2D` for a $k \times 800i$ array

Number of threads	ON	OFF	Speedup: ON Vs OFF
1	1.27	1.14	1.12
2 (scatter)	2.09	1.37	1.52
2 (compact)	2.16	1.65	1.31
4	3.00	1.84	1.63

(b) Impact of hardware prefetching for `memcpy2D` for $k \times 800i$ arrays

Number of threads	ON	OFF	Speedup: ON Vs OFF
1	1.89	1.86	1.02
2 (scatter)	2.69	2.11	1.28
2 (compact)	2.63	2.43	1.08
4	3.42	2.63	1.30

`memset2D` and `memcpy2D` `memset2D` is the simplest kernel presented in this paper. It sweeps a 2D array of double precision cells, and sets each of them to a given value. While it may look like some artificial-made code, similar code does exist in current commercial libraries. For example, Intel's Math Kernel Library (MKL) decomposes $C = A \times B$ in two steps: it first performs $C = 0$, then $C = 1 \times C + 1 \times A \times B$. This is also an interesting kernel, as it only performs stores to memory. Finally, such a "double precision 2D memset" is used in the MKL's DGEMM.

As Table 1(a) illustrates, turning the hardware prefetchers off always improves performance (up to 63%), no matter which configuration was chosen to pin down the threads.

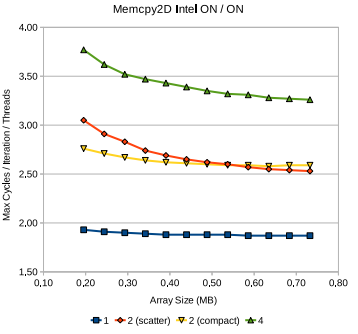
Because the array is partitioned along the second dimension (and not the first, see above), HW prefetchers load the next cacheline very aggressively *inside* an active thread's sibling dataset, effectively *stealing* useless cachelines for the sibling thread.

As for the `memcpy2D` kernel, a naive version of a 2D array copy is used [1].

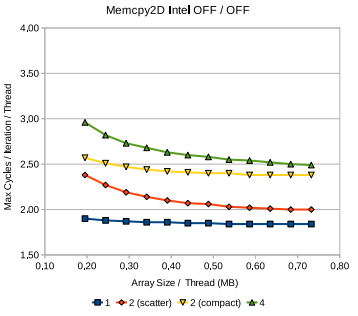
As before, Table 1(b) illustrates the performance gains which are always brought by turning prefetching off, whatever the threads configuration (up to 30% on four cores).

7-Point Stencil Operator. The 7-point stencil operator yields a certain amount of 3D arrays to load from, and a single array to store to. The geometry of the arrays was carefully chosen: the last dimension is far bigger than the first two, thus making it somewhat "logical" to divide work between threads along the third dimension. Indeed, this stencil operator accesses data in all directions of the arrays, and "cutting" them vertically (as shown in figure 1(b)) reduces the surface where data are to be shared between threads.

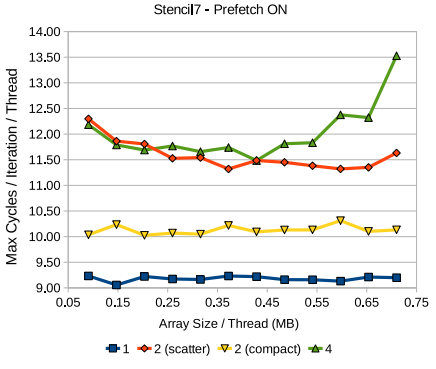
[1] It is worth mentioning that `icc` recognizes that the innermost loop of the naive version is an array copy and thus calls its own version of `memcpy` on a per-line basis.



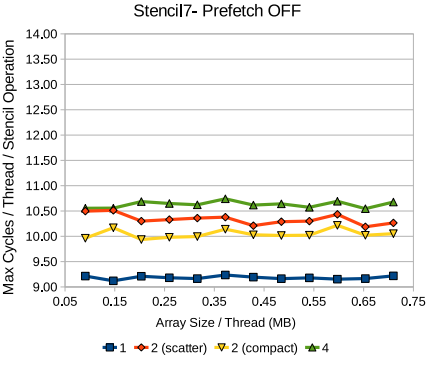
(a) Memcpy2D: Prefetch ON



(b) Memcpy2D: Prefetch OFF



(c) Stencil7: Prefetch ON



(d) Stencil7: Prefetch OFF

Fig. 2. The Impact of Cache-Line Stealing. On figure 2(a) (top left), where prefetching is ON, the amount of cycles per load-store is situated between 3.3 and 3.6 cycles on 4 cores, while on figure 2(b) (top right) where prefetching is OFF, the number of cycles continuously decreases to reach 2.5 cycles per load-store on 4 cores. On figure 2(c) (bottom left) prefetching is turned OFF, while on figure 2(d) prefetching is turned ON. "Scatter" and "compact" are two affinity policies defined in Intel's OpenMP runtime (see Section 2.2 for more details). Each thread processes the same amount of work. **Lower is better**.

Performance is very sensitive to the geometry of the data and the load-store ratio. For the 7-point stencil case, there is almost no advantage to turning off prefetching (except on 4-core) as there is a ratio of 14 loads for one store.

More "convenient" geometries were also used. 3D arrays would be shaped as $ik \times 10 \times 160$, with $3 \leq k \leq 32$, and i being the number of threads. Hence the size ratio between dimensions would become much less important. Data are distributed according to the storage policy, effectively making a thread access "its" subarray in a stride 1 manner. There is no CLS, but almost no advantage for using hardware prefetching either: speedups are flat (hence not shown here). Using such a geometry also shows that, contrary to the previous cases,

(c) Impact of hardware prefetching on the performance of `stencil7`. Shown are the average performance (in cycles per stencil computation step) and speedups. The baseline for speedups is the configuration where prefetching is turned ON. The dimensions of the 3D arrays are $k \times 10 \times 640i$, where $3 \leq k \leq 32$, and i is the number of threads running. 32-bits values are used in this example.

Threads #	ON	OFF	Speedup: OFF
1	9.22	9.23	1.00
2 (scatter)	11.60	10.38	1.12
2 (compact)	10.18	10.09	1.01
4	12.55	10.66	1.18

(d) Impact of hardware prefetching on the performance of `stencil7` in a favorable case. Shown are the average performance (in cycles per stencil computation step) and speedups. The baseline for speedups is the configuration where prefetching is turned ON.

Threads #	ON	OFF	Speedup: OFF
1	9.22	9.21	1.00
2 (scatter)	9.80	9.76	1.00
2 (compact)	9.78	9.75	1.00
4	10.14	10.23	0.99

whatever number of thread is used yields more or less the same performance on a per-thread basis. Thus contention is not the reason why in the previous cases performance decreases as the number of threads increases.

Table 1(d) sums up results for that geometry. Almost no difference appears, whatever configuration is chosen.

3 Possible Methods to Counter Cache-Line Stealing

CLS occurs mainly because data partitioning between threads was far from optimal. This is not necessarily trivial to avoid for the programmer, as such an ill-partitioned data set could happen when using external libraries on which the user has no control. In other cases there are inherent constrains to a given computation: for example when multiplying two matrices together, such as $C_{n,n} = A_{n,k} \times B_{k,n}$, where $k << n$ but both A and B still take a sizeable amount of space in memory, thus needing to block along not only on A, but also on B. It is first necessary to detect CLS. When it is clearly too important to ignore, then several countermeasures can be used against it.

Detecting Cache-Line Stealing. False-sharing can be detected using memory traces [9]. CLS, although different, might be tracked down the same way, tracking only stores.

Turning Off Prefetching. CLS can occur either in hardware or in software. Turning off prefetch is easy in a software context[2].

[2] For example, the `-no-prefetch` compiler option in Intel C Compiler (icc) enables the programmer to choose when not to activate software prefetching, which comes in handy for processors which support software prefetching only, such as the Itanium 2.

When talking about hardware prefetching, turning it off becomes more difficult, as it either requires physical access to the machine (which will need to be rebooted to access its BIOS or setup program), or because it requires superuser privileges to trigger it in software from the operating system. Used like this, prefetching can be turned off only when needed (i.e. potential cache-line stealing has been detected). The potential overhead of a per-process hardware prefetch trigger is not necessarily as high as the performance loss due to unnecessary data prefetching.

Loop Transformations. If an array is accessed along the wrong dimension (i.e. which leads to extensive CLS), it might simply be simpler to perform classical loop transformations on the loop nest which accesses it. For example, given a 2D array which was partitioned column-wise while storage policy is row-wise, loop interchange can totally void CLS, as is shown in figure 3.

On the x86 ISA, accessing the arrays column-wise hampers vector instruction generation, as there is no packed vector load (or store) memory instruction for non-unit strides. This effect could be worked around thanks to register blocking (i.e. accessing an array column-wise, but two cells by two cells). In this case the prefetcher will not be unnecessarily triggered.

Data Structure Reshaping. Finally, techniques similar to those used against false-sharing can be used (to a certain extent) against CLS. Array padding, while useful, will not prevent CLS as extra cache-lines are prefetched nonetheless. However array transposition, when possible, can help avoid CLS altogether, as it "reverses" the way cache-lines are packed together. For more complex data structures, techniques applied to avoid false-sharing can prove effective too.

This is more or less what was done in section 2.3, when arrays were reshaped to have a better way to parallelize the computation around them.

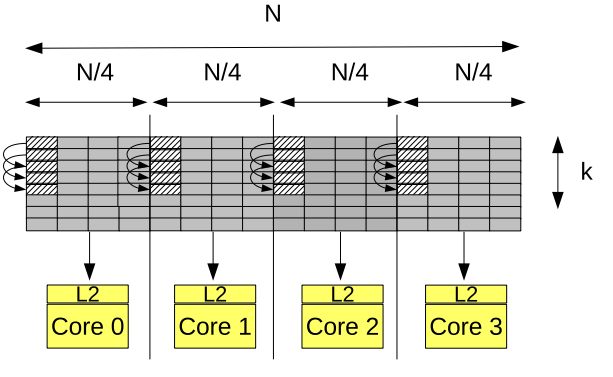


Fig. 3. A 2D array (stored row-wise) accessed column-wise by each thread. As data was distributed to threads by batches of columns rather than lines, accessing the array column-wise rather than row-wise does not prevent the IP-based prefetcher from fetching useful additional cache-lines (as there is a fixed stride), and avoids stealing adjacent cores' data.

4 Run-Time Adaptation to Solve Cache-Line Stealing Using a Hybrid Software/Hardware Framework

Cache-line stealing is not encountered very often, yet it can have a disastrous impact on performance, as section 2 shows. Finding out when CLS occurs is no easy task, as it both depends on the layout of the data structures, how data is partitioned between threads, and how many threads share arrays. Moreover, when using an external library, there is no certainty as to what a given library will do with the data types it takes as input or output. As the prefetch distance is seldom known in the case of hardware prefetchers (and as it can even be somewhat adaptive to the current workload of an application), the formula given in section 2 can only be applied with real success in the case of pure software prefetching. Other cases (i.e. hardware prefetching) must be dealt with differently.

We propose using adaptive methods such as the ones used by WEKA [6] to solve cache-line stealing, as very few different cases should occur in the kernels we are evaluating. Indeed, both `memset2D` and `memcpy2D`, as well as `stencil7` are "streaming" kernels, i.e. they are perfect loop nests with no control structure inside. Hence if performance is to vary, it is necessarily due to different input datasets. Moreover, adaptive methods can consider a given machine as some kind of "black box". Coupled with standard machine learning techniques, it is enough to spot problematic input datasets and try to generate a specific version of a kernel for similar cases.

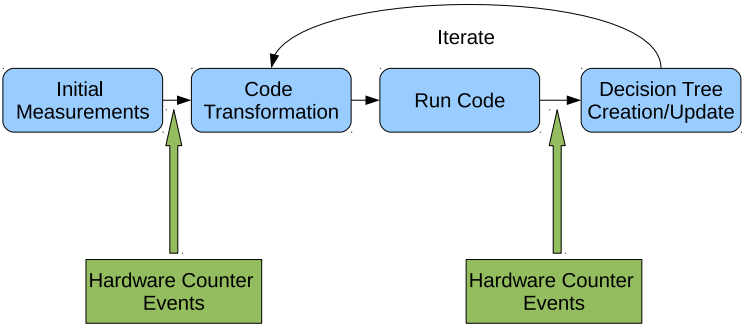
4.1 Description of the Adaptive Applications Framework

The framework used to counter CLS is explained in figure 4.

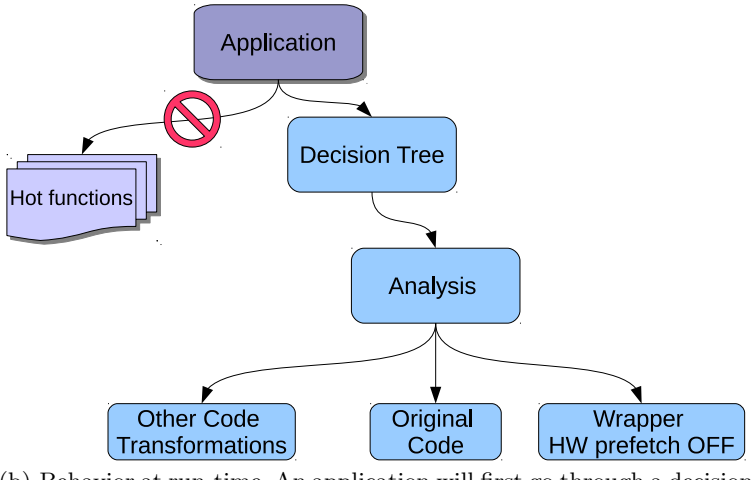
The hottest functions in a given application are identified during the initial measurements done on a given application. Then a driver is created to test such functions, to test its properties in a confined environment. In this specific case, only the shapes of the arrays given as input to the functions are of interest (be them 2D for `memset2D`/`memcpy2D` or 3D arrays). Each function is run to measure its IPC according to the various shapes the input arrays can have. IPCs are measured[3]. Usually, only a small number of shapes will incur a variation in the number of instructions per cycle. IPC is measured both for unicore and multicore executions, with various affinity policies. If an outstandingly bad IPC emerges from these tests, then a decision tree is generated or updated, which decides whether the original function or a wrapper (which turns off prefetching altogether for this specific input must) must be called. This wrapper function is also tested to make sure that its IPC is significantly better than the prefetch-enabled version. Figure 5 shows a C-like example code.

Once the good prefetch-less tuples {function, input dataset, partitioning scheme} are identified, a decision tree is generated or updated to switch on the right codelet to call at runtime. It must be noted that Figure 5 deals with

[3] Thanks to the `INSTRUCTION_RETIRED` and `CPU_CLK_UNHALTED` events.



(a) A given kernel is run with hardware counter events measurements. Code transformations are performed according to the results, and evaluated with HW counters. Finally, the decision tree is updated.



(b) Behavior at run-time. An application will first go through a decision tree which will quickly evaluate which kernel to use according to some evaluation of the input datasets.

Fig. 4. The adaptive compilation framework used to evaluate when to turn hardware prefetching off

```
return_type  wrapper_to_function_to_call(parameter1,  parameter2, ...)
{
    turn_off_prefetching();
    function_to_call(parameter1,parameter2, ...);
    turn_on_prefetching();
}
```

Fig. 5. Pseudo C-code for a wrapper to turn off prefetching for a given codelet

the case of hardware prefetchers. In the case of software prefetching (such as for the Itanium 2 processor), two possibilities exist:

1. All the source code is available. An exact copy of the function can thus be made, compiled with specific orders to turn prefetching off.
2. The function is located in a binary object. Then it must be directly modified, which is not currently possible. Additional features to manipulate binary executables must then be added to the framework.

On the Core 2 microarchitecture, there is no explicit user-space instruction to turn hardware prefetching on or off. Special files must be written to do so. Special rights must also be given to the user. In our different tests, trying to write too frequently into such a file[4] sometimes led to a system crash. Hence a more flexible mechanism to turn on or off hardware prefetching on different cores is a sorely missing feature in modern microprocessors.

Moreover, writing to such special files yields a severe overhead. There is a strong need for better hardware/software interactions. Such a hybrid approach with respect with data prefetching could actually perform better, as the resulting overhead would be much lower, as the few attempts at hybrid prefetching show [17,20,13].

4.2 Implementing the Adaptive Framework

Using IPC as an indicator, the framework was used on the `stencil7` kernel. Figure 6 illustrates the different shapes used for our experiments.

The decision tree thus built has two parameters: the shape of the 3D arrays, and the way data is partitioned. The latter can be inferred from the shape the arrays each thread has to process.

5 Related Work

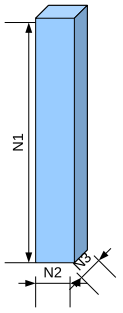
Hardware-based prefetching [5] eliminates the instruction overhead encountered in software-based prefetching [2,14] and may profit from runtime information, but tends to cause more unnecessary prefetches.

The problem of false-sharing has also been recognized for many years [4,21,10,3]. Data layout optimizations can help improve the memory performance of applications by controlling the way data is arranged in memory [16].

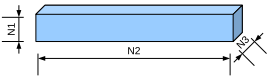
A benchmark enforcing false sharing with each write access to an array is introduced and analyzed for Intel Core Duo, Intel Xeon and AMD Opteron systems in [24]. The results show the impact of the cache architecture and of the coherency protocol on the performance.

Other benchmark results for various platforms on sparse matrix-vector product [26] show the impact of prefetching on such operations. Another example can be found in [25], not directly referring to cache-line stealing, but systematically benchmarking the impact of enabled/disabled hardware and software

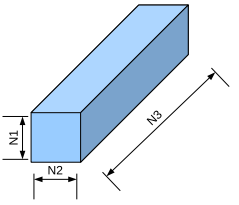
[4] Under Linux, we are talking of the `/dev/msr/*` files.



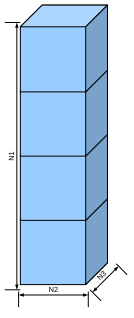
(a) Narrow case: $N_1 >> (N_2, N_3)$. Only the outermost ("outer" case) loop is parallelized.



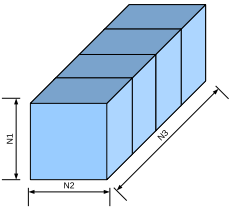
(b) Narrow case: $N_2 >> (N_1, N_3)$. Either the outermost ("outer" case) or the innermost ("inner" case) loop is parallelized.



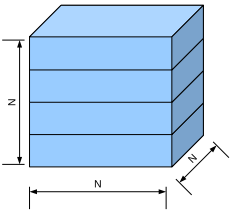
(c) Narrow case: $N_3 >> (N_1, N_2)$. Only the innermost ("inner" case) loop is parallelized.



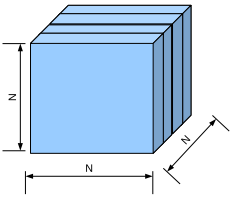
(d) Each thread processes a cube: "outer" case.



(e) Each thread processes a cube: "inner" case.



(f) The whole arrays are cubes: each thread processes a hyperplan ("outer" case).



(g) The whole arrays are cubes: each thread processes a hyperplan ("inner" case).

Fig. 6. Various 3D array shapes tested for their IPC in the case of the `stencil7` kernel

data prefetching on a medical imaging application running on Intel dual-core processors.

The notion of memory access intensity is introduced in [12] to facilitate quantitative analysis of a program's memory behavior on multicore systems with hardware prefetching and is demonstrated with three numerical solvers for large scale sparse linear systems.

Some research with different solutions have been proposed to solve the problem of prefetching in a multicore/multiprocessor context [11,22].

Finally, hybrid software/hardware data prefetching has been the interest of too few researchers. Wang et al [23] propose to improve on scheduled region prefetching (SRP) with guided region prefetching (GRP). Although the results are promising, the paper focuses on single-core execution only. Seung Woo Son et al [18] show how conventional data prefetching techniques usually don't scale on CMP systems. They propose compiler-driven prefetching techniques, which they validate on a simulator. To conclude, Gornish and Veidenbaum [7] propose a hybrid mechanism, which consists in using software prefetching before a given loop in order to "train" the hardware prefetchers, informing them of the stride to use, as well as the when to stop prefetching. This mechanism, if implemented, could really bring some leverage to the programmer and compiler writer to efficiently bring data back into the right cache, for the right core.

To our best knowledge, there has not been much published on investigating the impact of prefetching on "cache-line stealing" on contemporary multicore architectures, for either software- or hardware-based prefetching techniques. Close to that topic, the work of Song et al [19] tries to make an accurate model of direct-mapped caches to predict compulsory misses, capacity misses (both on private and shared data), and when cache hits become cache misses as well as the contrary.

6 Conclusion and Future Work

While very useful, data prefetching can severely hurt performance by triggering cacheline stealing. While the unicore case has been studied in depth, the multicore one features almost no study. Hardware prefetchers do not give the opportunity to the user to choose the prefetch distance according to some known pattern. Hence if cache-thrashing occurs, it is unpreventable. On the software-side, the shape of the arrays still conditions the efficiency of software prefetching, and this can only be corrected through iterative methods. Otherwise, cacheline stealing may occur.

Cache-line stealing induced by too aggressive data prefetching was studied in this paper, along with a way to compute the number of extra cache-lines brought into the cache-hierarchy of a given core for a given prefetch distance – which unfortunately can only be empirically guessed in the case of hardware prefetchers. Prefetching coupled with a certain way of partitioning data between threads provokes the apparition of cache-line stealing.

Some solutions were proposed to eliminate some or all of cache-line stealing, such as

- Turning off prefetching. A potential solution could come from microprocessor vendors to allow some kind of per-process way of activating or deactivating prefetching.
- Performing loop transformations on the incriminated code.
- Reshaping the data structures to better suit the prefetching policy.

Finally, an adaptive framework was proposed to counter cache-line stealing by deactivating hardware prefetching on Core 2 processors only when necessary.

Although this solution should provide a perfect hybrid way to solve CLS, the current way to turn HW prefetching on or of offsets the gains. We strongly advocate for a better hardware/software interaction in the case of prefetching, to reduce some of the negative impact hardware prefetching can have on performance in multicore.

Future work includes refining this adaptive framework to include other, more complex transformations. Among them, the ones described in section 3, which were performed by hand so far.

Acknowledgments

The research presented in this paper is supported by ITACA LAB (University of Versailles and CEA-DAM common lab) and the ITEA2 project "ParMA" [1] (June 2007 – May 2010).

References

1. ParMA: Parallel programming for multi-core architectures - ITEA2 project (06015), http://www.parma-itea2.org
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler Transformations for High-Performance Computing. ACM Comput. Surv. 26(4), 345–420 (1994)
3. Bodin, F., Granston, E.D., Montaut, T.: Evaluating two loop transformations for reducing multiple writer false sharing. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) LCPC 1994. LNCS, vol. 892, pp. 421–439. Springer, Heidelberg (1995)
4. Bolosky, W.J., Scott, M.L.: False Sharing and its effect on shared memory performance. In: Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), pp. 57–71 (1993)
5. Garzaran, M., Brit, J., Ibanez, P., Vinals, V.: Hardware Prefetching in Bus-Based Multiprocessors: Pattern Characterization and Cost-Effective Hardware. In: Ninth Euromicro Workshop on Parallel and Distributed Processing, pp. 345–354 (2001)
6. Holmes, G., Donkin, A., Witten, I.: WEKA: a machine learning workbench. In: Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems, pp. 357–361 (1994)
7. Gornish, E.H., Veidenbaum, A.: An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. Intl. Journal of Parallel Programming, 35–70 (1999)
8. Hedge, R.: Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers (2008), http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers/
9. Hyde, R.L., Fleisch, B.D.: An analysis of degenerate sharing and false coherence. J. Parallel Distrib. Comput. 34(2), 183–195 (1996)
10. Jeremiassen, T.E., Eggers, S.J.: Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In: PPOPP, pp. 179–188 (1995)

11. Jerger, N., Hill, E., Lipasti, M.: Friendly fire: understanding the effects of multiprocessor prefetches. In: IEEE International Symmposium on Performance Analysis of Systems and Software, pp. 177–188 (2006)
12. Liu, L., Li, Z., Sameh, A.H.: Analyzing memory access intensity in parallel programs on multicore. In: ICS 2008, pp. 359–367. ACM, New York (2008)
13. Marathe, J., Mueller, F., de Supinski, B.R.: Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. ACM Trans. Archit. Code Optim. 3(4), 390–423 (2006)
14. Mowry, T.C.: Tolerating Latency in Multiprocessors Through Compiler-Inserted Prefetching. ACM Trans. Comput. Syst. 16(1), 55–92 (1998)
15. Papamarcos, M.S., Patel, J.H.: A low-overhead coherence solution for multiprocessors with private cache memories. In: ISCA 1984: Proceedings of the 11th Annual International Symposium on Computer Architecture, pp. 348–354. ACM, New York (1984)
16. Raman, E., Hundt, R., Mannarswamy, S.: Structure Layout Optimization for Multithreaded Programs. In: CGO, pp. 271–282. IEEE Computer Society, Los Alamitos (2007)
17. Skeppstedt, J., Dubois, M.: Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. In: International Conference on Parallel Processing, p. 298 (1997)
18. Son, S.W., Kandemir, M., Karakoy, M., Chakrabarti, D.: A compiler-directed data prefetching scheme for chip multiprocessors. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 209–218. ACM, New York (2009)
19. Song, F., Moore, S., Dongarra, J.: L2 cache modeling for scientific applications on chip multi-processors. In: ICPP 2007: Proceedings of the 2007 International Conference on Parallel Processing, Washington, DC, USA, p. 51. IEEE Computer Society, Los Alamitos (2007)
20. Struik, P., van der Wolf, P., Pimentel, A.D.: A combined hardware/software solution for stream prefetching in multimedia applications (1998)
21. Torrellas, J., Lam, M.S., Hennessy, J.L.: False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers 43, 651–663 (1994)
22. Wallin, D., Hagersten, E.: Miss penalty reduction using bundled capacity prefetching in multiprocessors. In: International Parallel and Distributed Processing Symposium, p. 12a (2003)
23. Wang, Z., Burger, D., McKinley, K.S., Reinhardt, S.K., Weems, C.C.: Guided region prefetching: A cooperative hardware/software approach. In: Proceedings of the 30th International Symposium on Computer Architecture, pp. 388–398 (2003)
24. Weidendorfer, J., Ott, M., Klug, T., Trinitis, C.: Latencies of Conflicting Writes on Contemporary Multicore Architectures. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 318–327. Springer, Heidelberg (2007)
25. Whitepaper, I.: Optimizing Embedded System Performance - Impact of Data Prefetching on a Medical Imaging Application (2006), `http://download.intel.com/technology/advanced_comm/315697.pdf`
26. Williams, S., Oliker, L., Vuduc, R.W., Shalf, J., Yelick, K.A., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: SC 2007, p. 38 (2007)
27. Wulf, W.A., McKee, S.A.: Hitting the Memory Wall: Implications of the Obvious. Computer Architecture News 23, 20–24 (1995)