Platform-Independent Cache Optimization by Pinpointing Low-Locality Reuse

Kristof Beyls and Erik H. D'Hollander

Departement of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{kristof.beyls,erik.dhollander}@elis.ugent.be

Abstract. For many applications, cache misses are the primary performance bottleneck. Even though much research has been performed on automatically optimizing cache behavior at the hardware and the compiler level, many program executions remain dominated by cache misses. Therefore, we propose to let the programmer optimize, who has a better high-level program overview, needed to resolve many cache problems. In order to assist the programmer, a visualization of memory accesses with poor locality is developed. The aim is to indicate causes of cache misses independent of actual cache parameters such as associativity or size. In that way, the programmer is steered towards platform-independent locality optimizations. The visualization was applied to three programs from the SPEC2000 benchmarks. After optimizing the source code based on the visualization, an average speedup of 3.06 was obtained on different platforms with Athlon, Itanium and Alpha processors; indicating the feasibility of platform-independent cache optimizations.

1 Introduction

On current processors, the execution time of many programs is dominated by processor stall time during cache misses. In figure 1, the execution time of the SPEC2000 programs is categorized into data cache miss, instruction cache miss, branch misprediction and useful execution. On average, almost 50% of the execution time, the processor is stalled due to data cache misses.

Several studies on different benchmark suites have shown that *capacity misses* are the dominant category of misses[7,2,8,4]. Therefore, we focus on reducing capacity misses.

Cache misses can generally be resolved at three different levels: the hardware level, the compiler level or the algorithm level. At the hardware level, capacity misses can only be eliminated by increasing the cache size[7], which makes it slower. Previous research indicates that state-of-the-art compiler optimizations can only reduce about 1% of all capacity misses[2]. This shows that capacity miss optimization is hard to automatize, and should be performed by the programmer. However, cache behavior is not obvious from the source code, so a tool is needed to help the programmer pin-point the causes of cache bottlenecks. In contrast

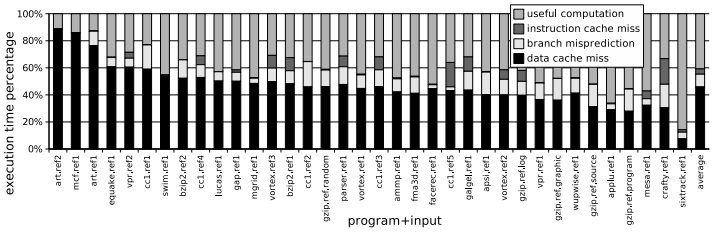


Fig. 1. Breakdown of execution time for the SPEC2000 programs, on a 733Mhz Itanium1 processor. The programs are compiled with Intel's compiler, using the highest level of feedback-driven optimization.

to earlier tools[1,3,6,10,11,12,14,15], we aim at steering the programmer towards platform-independent cache optimizations.

The key observation is that capacity misses are caused by low-locality accesses to data that has been used before. Since the data has already been accessed before, it has been in the cache before. However, due to low locality, it's highly unlikely that the cache retains the data until the reuse. By indicating those reuses, the programmer is pointed to the accesses where cache misses can be eliminated by improving locality.

Both temporal and spatial locality are quantified, by locality metrics, as discussed in section 2. Section 3 presents how the metrics are graphically presented to the programmer. The visualization has been applied to the three programs in SPEC2000 with the largest cache miss bottleneck: `art`, `mcf` and `equake` (see fig. 1). A number of rather small source code optimizations were performed, which resulted in platform-independent improvements. The average speedup of these programs on Athlon, Alpha and Itanium processors was 3.06, with a maximum of 11.23. This is discussed in more detail in section 4. In section 5, concluding remarks follow.

2 Locality Metrics

Capacity misses are generated by reuses with low locality. Temporal locality is measured by *reuse distance*, and spatial locality is measured by *cache line utilization*.

Definition 1. *A* **memory reference** *corresponds to a read or a write in the source code, while a particular execution of that read or write at runtime is a* **memory access**. *A* **reuse pair** $\langle a_1, a_2 \rangle$ *is a pair of accesses in a memory access stream, which touch the same location, without intermediate accesses to that location. The* **reuse distance** *of a reuse pair* $\langle a_1, a_2 \rangle$ *is the number of unique memory locations accessed between* a_1 *and* a_2. *A* **memory line** *is a cache-line-sized block of contiguous memory, containing the bytes that are mapped into a single cache line. The* **memory line utilization** *of an access a to memory line l is the fraction of l which is used before l is evicted from the cache.* □

Lemma 1. *In a fully associative LRU cache with n lines, a reuse pair $\langle a_1, a_2 \rangle$ with reuse distance $d < n$ will generate a hit at access a_2. If $d \geq n$, a_2 misses the cache.[2]* □

The accesses with reuse distance $d \geq n$ generate capacity misses. When the backward reuse distance of access a is larger than the cache size, a cache miss results and memory line l is fetched into the cache. If the memory line utilization of a is less than 100%, not all the bytes in l are used, during that stay in the cache. Therefore, at access a, some useless bytes in l were fetched, and the potential benefit of fetching a complete memory line was not fully exploited. The memory line utilization shows how much spatial locality can be improved.

3 Measurement and Visualization

3.1 Instrumentation and Locality Measurement

In order to measure the reuse distance and the memory line utilization, the program is instrumented to obtain the memory access trace. The ORC-compiler[13] was extended, so that for every memory access instruction, a function call is inserted. The accessed memory address, the size of the accessed data and the instruction generating the memory access are given to the function as parameters. The instrumented program is linked with a library that implements the function which calculates the reuse distance and the memory line utilization. The reuse distance is calculated in constant time, in a similar way as described in [9]. For every pair of instructions (r_1, r_2), the distribution of the reuse distance of reuse pairs $\langle a_1, a_2 \rangle$ where a_1 is generated by r_1 and a_2 is generated by r_2 is recorded.

The memory line utilization of a reuse pair is measured by choosing a fixed size $\mathsf{CS}_{\mathsf{min}}$ as the minimal cache size of interest. On every access l, it is recorded which bytes were used. When line l is evicted from the cache of size $\mathsf{CS}_{\mathsf{min}}$, the fraction of bytes accessed during that stay of l in cache of size $\mathsf{CS}_{\mathsf{min}}$ is recorded. In the experiments in section 4, $\mathsf{CS}_{\mathsf{min}} = 2^8$ cache lines of 64 bytes each = 16 KB. At the end of the program execution, the recorded reuse distance distributions, together with the average memory line utilizations are written to disk.

A large overhead can result from instrumenting every memory access. The overhead is reduced by sampling, so that reuses are measured in bursts of 20 million consecutive accesses, while the next 180 million accesses are skipped. In our experiments, a slowdown between 15 and 25 was measyred The instrumented binary consumes about twice as much memory as the original, due to bookkeeping needed for reuse distance calculation.

3.2 Visualization

Lemma 1 indicates that only those reuses whose distance is larger than the cache size generate capacity misses. Therefore, only the reuse pairs with a long reuse distance (=low locality) are shown to the programmer. Furthermore, in order to guide the programmer to the most important low-locality reuse places in the

program, only the instructions pairs generating at least 1% of the long distances are visualized. The visualization shows these pairs in as arrows drawn on top of the source code. A label next to the arrow shows how many percent of the long reuse distances were generated by that reuse pair. Furthermore, the label also indicates the memory line utilization of the cache missing accesses generated by that reuse pair. A simple example is shown in fig. 2.

3.3 Program Optimization

The locality measurement and visualization is performed automatically. Based on the visualization, the capacity misses can be reduced by the programmer. There are four basic ways in which the capacity misses, or their slowdown effect, can be reduced.

1. *Eliminate the memory accesses with poor locality* altogether.
2. Reduce the distance between use and reuse for long reuse distances, so that it becomes smaller than the cache size. This can be done by reordering computations (and memory accesses), so that *higher temporal locality* is achieved.
3. *Increase the spatial locality* of accesses with poor spatial locality. This is most easily done by rearranging the data layout.
4. If neither of the three previous methods are applicable, it might still be possible to improve the program execution speed by *hiding their latency with independent parallel computations*. The most well-known technique in this class is prefetching.

4 Experiments

In order to evaluate the benefit of visualizing low-locality reuses, the three programs from SPEC2000 with the high cache bottlenecks were considered: 181.mcf, 179.art and 183.equake (see fig. 1). Below, for each of the programs, the visualization of the main cache bottlenecks is shown. Also, it is discussed how the programs were optimized.

4.1 Mcf

The main long reuse distances for the 181.mcf program are shown in figure 2. The figure shows that about 68% of the capacity misses are generated by a single load instruction on line 187. The best way to solve those capacity misses would be to shorten the distance between the use and the reuse. However, the reuses of `arc`-objects occur between different invocations of the displayed function. So, bringing use and reuse together needs a thorough understanding of the complete program, which we do not have, since we didn't write the program ourselves. A second way would be to increase the spatial locality from 21% to a higher percentage. In order to optimize the spatial locality, the order of the fields of the `arc`-objects could be rearranged. However, this change leads to poorer spatial locality in other parts of the program, and overall, this restructuring does not lead

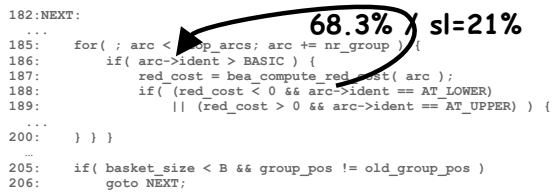


Fig. 2. A zoom in on the major long reuse distance in 181.mcf. A single pair of instructions produce 68.3% of the long reuse pairs. The second number (sl=21%) indicates that the cache-missing instruction (on line 186) has a memory line utilization of 21%.

to speedup. Therefore, we tried the fourth way to improve cache performance: inserting prefetch instructions to hide the miss penalty.

4.2 Art

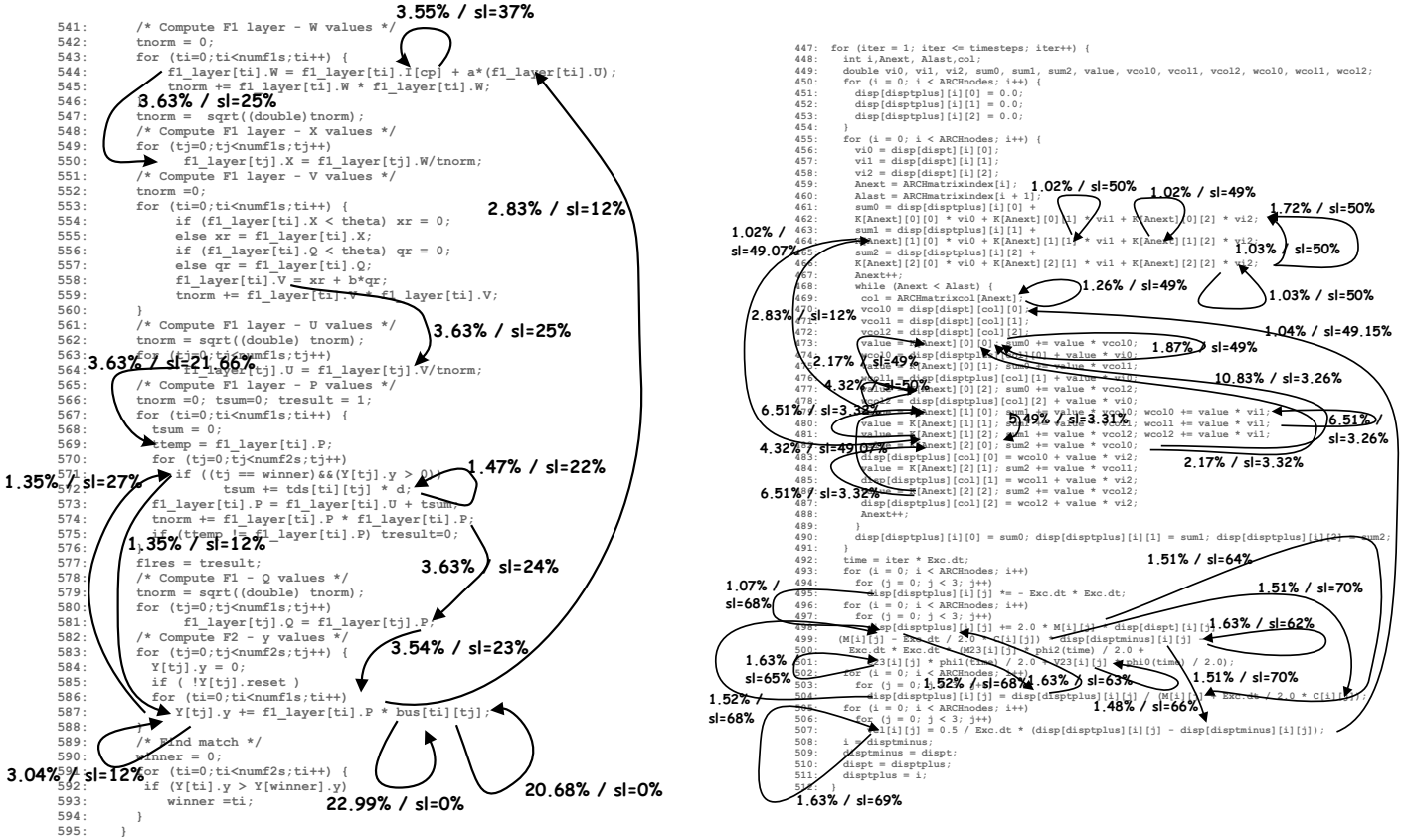
The 179.art program performs image recognition by using a neural network. A bird's-eye view of the main long reuse distances in this program is shown in figure 3(a). Each node in the neural network is represented by a struct containing 9 `double`-fields, which are layed out in consecutive locations. The visualization shows low spatial locality. The code consists of 8 loops, each iterating over all neurons, but only accessing a small part of the 9 fields in each neuron. A simple data layout optimization resolves the spatial locality problems. Instead of storing complete neurons in a large array, i.e. an array of structures, the same field for all the neurons are stored consecutively in arrays, i.e. a structure of arrays. Besides this data layout optimization, also some of the 8 loops were fused, when the data dependencies allowed it and reuse distances were shortened by it.

4.3 Equake

The equake program simulates the propagation of elastic waves during an earthquake. A bird's-eye view on the major long reuse distance in this program is shown in figure 3(b). All the long reuse distances occur in the main simulation loop of the program which has the following structure:

- Loop for every time step (line 447–512):
  - Loop to perform a sparse matrix-vector multiplication. (line 455–491)
  - A number of loops to rescale a number of vectors. (line 493–507)

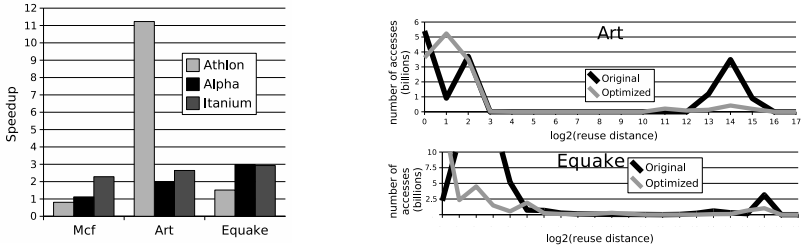
Most of the long reuse distances occur in the sparse matrix-vector multiplication, for the accesses to the sparse matrix `K`, declared as `double*** K`. The matrix is symmetric, and only the upper triangle is stored. An access to an element has the form `K[Anext][i][j]`, leading to three loads. The number of memory accesses are reduced by redefining `K` as a single dimensional array and replacing



(a) art

(b) equake

Fig. 3. A zoom-out view of the major long reuse distances in 179.art and 183.equake.



(a) Speedups on different architectures

(b) Reuse distance distribution

Fig. 4. (a) Speedups and (b) Reuse distance distributions before and after optimization.

the accesses into `K[Anext*N*9+i*3+j]`, leading to a single load instruction. Furthermore, after analyzing the code a little further, it shows that for most of the long reuse pairs, the use is in a given iteration of the time step loop and the reuse is in the next iteration. In order to bring the use and the reuse closer together, some kind of tiling transformation should be performed on the time-step loop (i.e. try to do computations for a number of consecutive time-steps on the set of array elements that are currently in the cache). All the sparse matrix elements were stored in memory, and not only the upper triangle, which allows to simplify the sparse code and remove some loop dependences. After this, it was possible to fuse the matrix-vector multiply loop with the vector rescaling loops, resulting in a single perfectly-nested loop. In order to tile this perfectly-nested loop, the structure of the sparse matrix needs to be taken into account, to figure out the real dependencies, which are only known at run-time. The technique described in [5] was used to perform a run-time calculation of a legal tiling.

4.4 Discussion

The original and optimized programs were compiled and executed on different platforms: an Athlon PC, an Alpha workstation and an Itanium server. For the Athlon and the Itanium, the Intel compiler was used. For the Alpha 21264, the Alpha compiler was used. All programs were compiled with the highest level of feedback-driven optimization. In figure 4(a), the speedups are presented, showing that most programs have a good speedup on most processors. Figure 3(b) shows that the long reuse distances have been effectively diminished in both art and equake. In mcf (not displayed), the reuse distances were not diminished, since only prefetching was applied. Only for mcf on the Athlon, a slow-down is measured, probably because the hardware-prefetcher in the Athlon interferes with the software prefetching.

5 Conclusion

On current processors, on average about half of the execution time is caused by cache misses, after applying hardware and compiler optimizations, making it the number one performance bottleneck. Therefore, the programmer needs to improve the cache behavior. A method is proposed which measures and visualizes the reuses with poor locality, resulting in cache misses. The visualization hints the programmer into portable cache optimizations.

The effectiveness of the visualization was tested by applying it to the three programs with the highest cache bottleneck in SPEC2000. Based on the visualization, a number of rather simple source code optimizations were applied. The speedups of the optimized programs were measured on CISC, RISC and EPIC processors, with different underlying cache architectures. Even after full compiler optimization, the visualization enabled the programmer to attain speedups of up to 11.23 on a single processor (see fig. 4). Furthermore, the applied optimizations lead to a consistent speedup in all but one case.

References

1. E. Berg and E. Hagersten. Sip: Performance tuning through source code interdependence. In *Euro-Par'02*, pages 177–186, 2002.
2. K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, Aug 2001.
3. R. Bosch and C. S. et al. Rivet: A flexible environment for computer systems visualization. *Computer Graphics-US*, 34(1):68–73, Feb. 2000.
4. J. F. Cantin and M. D. Hill. Cache performance for selected SPEC CPU2000 benchmarks. *Computer Architecture News (CAN)*, September 2001.
5. C. C. Douglas and J. H. et al. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal.*, 10:25–40, 2000.
6. A. Goldberg and J. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, 1993.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2002.
8. M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.
9. Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *ACM SIGMETRICS conference*, pages 212–213, 1991.
10. A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.
11. M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance in sequential and parallel programs. *IEEE Computer*, April 1995.
12. J. Mellor-Crummey and R. F. et al. HPCView: a tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–104, 2002.
13. Open research compiler. http://sourceforge.net/projects/ipf-orc.
14. E. Vanderdeijl, O. Temam, E. Granston, and G. Kanbier. The cache visualization tool. *IEEE Computer*, 30(7):71, 1997.
15. Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of cache on the program execution. In *Information Visualization 2001*, pages 336–341, 2001.