Reuse Distance as a Metric for Cache Behavior

Kristof Beyls Erik H. D'Hollander
Ghent University
Sint-Pietersnieuwstraat 41,
9000 Ghent, Belgium
(email: {kristof.beyls,erik.dhollander}@elis.rug.ac.be)

ABSTRACT
The widening gap between memory and processor speed causes more and more programs to shift from CPU-bounded to memory speed-bounded, even in the presence of multi-level caches. Powerful cache optimizations are needed to improve the cache behavior and increase the execution speed of these programs. Many optimizations have been proposed, and one can wonder what new optimizations should focus on. To answer this question, the distribution of the conflict and capacity misses was measured in the execution of code generated by a state-of-the-art EPIC compiler. The results show that cache conflict misses are reduced, but only a small fraction of the large number of capacity misses are eliminated. Furthermore, it is observed that some program transformations to enhance the parallelism may counter the optimizations to reduce the capacity misses. In order to minimize the capacity misses, the effect of program transformations and hardware solutions are explored and examples show that a directed approach can be very effective.

KEY WORDS
processor cache, reuse distance, stack distance, capacity misses

1. Introduction

With the increasing divergence between processor and memory speed, caches play a key role in minimizing the data access latency and main memory bandwidth demand. However, caches are not perfect and many techniques have been proposed to reduce the number of cache misses, both at the hardware and at the software level. Most of the proposed methods focus on eliminating conflict misses[3, 5, 9, 11, 13, 15]. Only some software techniques such as loop tiling, loop fusion and loop reversal focus on eliminating capacity misses. However, in real programs, such as spec95fp, it has been shown that up to 68% of the cache misses are capacity misses[8]. This is consistent with our own measurements, presented in section 3..

In order to gain insight in the origin of cache misses, we propose to measure the stack distance[2] as a metric for data locality in sect. 2.. It measures the distance in time between the use and subsequent reuse of the same data location. Section 3. shows that the stack distance predicts the number of cache misses accurately. The performance of current software optimization techniques was measured, using an EPIC (Explicitly Parallel Computing)[12] compiler. The compiler incorporates many high-level program optimizations, steered towards increasing the instruction level parallelism and the data locality. Both are important in targeting an EPIC machine, since the parallelism needs to be expressed explicitly in the generated code, taking into account the cache behavior. It shows that the existing optimizations reduce conflict misses more easily than capacity misses. An overview of capacity miss reducing optimizations and their effect on the stack distance is described in section 4.. However, code optimizations to increase fine grain parallelism can increase the number of cache misses. Concluding remarks are given in section 5..

2. Stack Distance as a Reuse Metric

2.1 Definitions

Caches exploit data reuse. The 3C's(cold, conflict, capacity) model[4] categorizes cache misses according to the cause of a cache miss. A *cold miss* occurs when data is referenced for the first time. *Capacity misses* occur when the requested data has been expelled because too much different data has entered the cache since the last reference: the "reuse distance" is too large. *Conflict misses* occur when the requested data has been expelled because too much different data has entered the associated cache lines. In order to measure data reuse, we propose the stack distance[2] as a metric.

Definition 1 *The* stack distance *of a memory access is the number of accesses to* unique *addresses made since the last reference to the requested data.* □

The stack distance can be computed as follows. Consider the address trace of a program. Iterate over the data address trace sequentially, and place the address under consideration onto a stack. If the address was referenced before, move it to the top of the stack. The stack distance is the depth of the address in the stack when it is referenced. It has the following interesting property:

Lemma 1 *In a fully associative LRU cache with n lines, a reference with stack distance $d < n$ will hit. A reference*

reference no.	1	2	3	4	5	6	7
address	0	8	16	96	8	16	104
corresponding cache line	0	0	1	6	0	1	6
stack distance (address gran.)	∞	∞	∞	∞	2	2	∞
stack distance (line gran.)	∞	0	∞	∞	2	2	2

Figure 1. An example of stack distance measured by address and cache line granularity respectively, for a cache line size of 16 bytes. In the latter case, spatial locality is captured by the stack distance.

with stack distance $d \geq n$ will miss.

Proof *In a fully-associative LRU cache with n lines, the n most recently referenced cache lines are retained. When a reference has stack distance d, exactly d different cache lines were referenced previously. If $d \geq n$, the referenced cache line is not one of the n most recently referenced cache lines, and consequently will not be found in the cache.* □

As a consequence of this property, the classification of a miss into a cold, conflict or capacity misses is easily made using the stack distance. A cold miss has an infinite stack distance, since it was not previously referenced. If the stack distance is smaller than the number of cache lines, it is a conflict miss, since the same reference would have been a hit in the fully associative cache[4]. When the stack distance is larger than the number of cache lines, it is a capacity miss, since the reference also misses in a fully-associative cache.

The reuse distance quantifies *temporal locality*. However, *spatial locality* also needs to be represented in the stack distance. We measure the stack distance with cache line granularity instead of data element granularity. In this way the number of unique cache lines accessed since the last access to the current cache line is measured. As a result, the spatial locality exploited by the cache is measured. An example is shown in figure 1.

Previously, "reference distance" has been proposed as a metric for data locality[10]. The reference distance is the *total* number of reference between accesses to the same data. In fig. 2, the difference between reference distance and stack distance is illustrated. The reference distance cannot exactly predict the cache behavior for a cache, whereas the stack distance can do it for fully associative caches.

2.2 Efficient Stack Distance Measurement

The parallelizing compiler SUIF[16] was extended to instrument Fortran programs to obtain the memory reference trace. The resulting trace is fed into a stack distance analyzer.

The stack distance analyzer measures the stack distance efficiently by storing the accessed memory lines in a balanced binary tree. It allows to calculate the stack distance in $O(\log d)$ time, where d is the number of different memory lines accessed during the program execution. The nodes in the tree are ordered. Nodes in the left subtree are more recently referenced, and nodes in the right subtree are less recently referenced. In every node, the number of nodes in its left subtree is remembered. This allows to quickly calculate the number of unique memory lines that were accessed before the memory line under consideration.

In previous related work[10], the reference distance instead of the stack distance was measured for sake of efficiency. However, with the above efficient implementation, measuring the stack distance took us only 40% more time than measuring the reference distance. With a bit more processing time, the more accurate stack distance can be measured.

3. Program Analysis

The stack distance analyzer was used to measure the distance distributions for the references, hits and misses of both original and optimized versions of the SPEC95 floating point programs. The simulated cache is a 32 KB direct mapped cache with 32 bytes line size. So, the stack distances are all measured with a granularity of 32-bytes and the misses at a stack distance exceeding 1024 are capacity misses.

3.1 Cache Behavior before Optimization

The 10 programs in the SPEC95fp benchmark suite were instrumented to obtain the distribution of hits and misses in a 32KB direct mapped cache. On average, 32% of the misses are conflict misses, while 68% are capacity misses. So, even for a direct mapped cache, where many conflicts are expected, only a minority of the misses are conflict misses.

The hit percentage for references categorized by stack distance, both for the direct mapped and a fully associative cache, is plotted in figure 3. From the figure, it can be seen that the stack distance is a good predictor for cache behavior, even for a direct mapped cache. For a stack distance smaller than the cache size (number of cache lines is $1024 = 2^{10}$), the hit percentages are very high. When the stack distance is larger than the number of cache lines, the hit percentage drops to 0%. For a fully-associative cache, the plot shows no misses for references smaller than the cache size (in cache lines), and 100% misses for larger stack distances. With a decreasing associativity, the cache behavior of the references will diverge from the cache behavior of the references in a fully associative cache. For a direct mapped cache the hit rates decrease a bit more gradu-

reference number	1	2	3	4	5	6	7	8	9	10	11	12
memory references	A	B	B	A	C	A	B	B	B	A	B	C
reference distance	∞	∞	0	2	∞	1	3	0	0	3	1	6
stack distance	∞	∞	0	1	∞	1	2	0	0	1	1	2
hit in cache with 2 cache lines?(sd$<$ 2)	n	n	y	y	n	y	n	y	y	y	y	n

Figure 2. The reference and stack distance for a short memory trace. The reference hits in a fully-associative cache with 2 cache lines if the stack distance is smaller than 2. When the reference distance is smaller than the cache size, the reference results in a hit. When the reference distance is larger, no conclusion about the cache behavior of the reference can be made. For example, reference 10 has reference distance 3 (larger than the cache size), but it is a hit. On the other hand, using the stack distance of 1, it can be concluded that it is a hit.

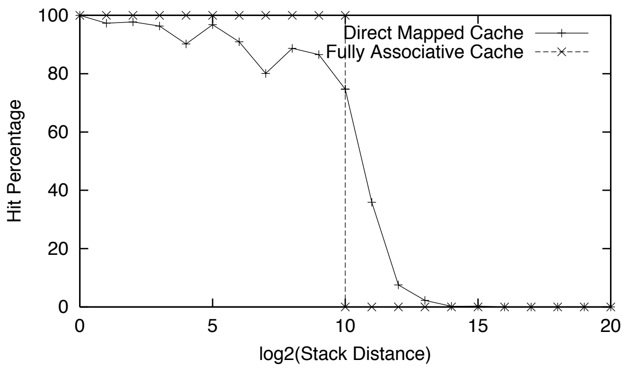

Figure 3. The average hit percentage of the spec95fp benchmark vs. the stack distance, both for a direct mapped and a fully associative cache.

ally for increasing stack distances. However, even for direct mapped caches, the hit rate curves have a sharp drop around the cache size. For set-associative caches, the hit percentage plot will be in between that of the direct mapped and the fully associative cache. So, the stack distance predicts the cache behavior accurately, even in the case of a direct mapped cache.

3.2 Effect of Compiler Optimization on Cache Misses

The previous section shows that about 70% of the cache misses are capacity misses. Many hardware and software optimizations have been proposed to improve the cache behavior of programs. However most of these concentrate on resolving conflict misses, leaving only loop fusion, loop tiling and loop reversal to minimize capacity misses. In an attempt to measure the effectiveness of existing software optimizations in eliminating capacity misses, the SGI Pro64 compiler[14], which generates IA-64 EPIC (Explicitly Parallel Computing) code, was used. The compiler incorporates many high-level program optimizations, such

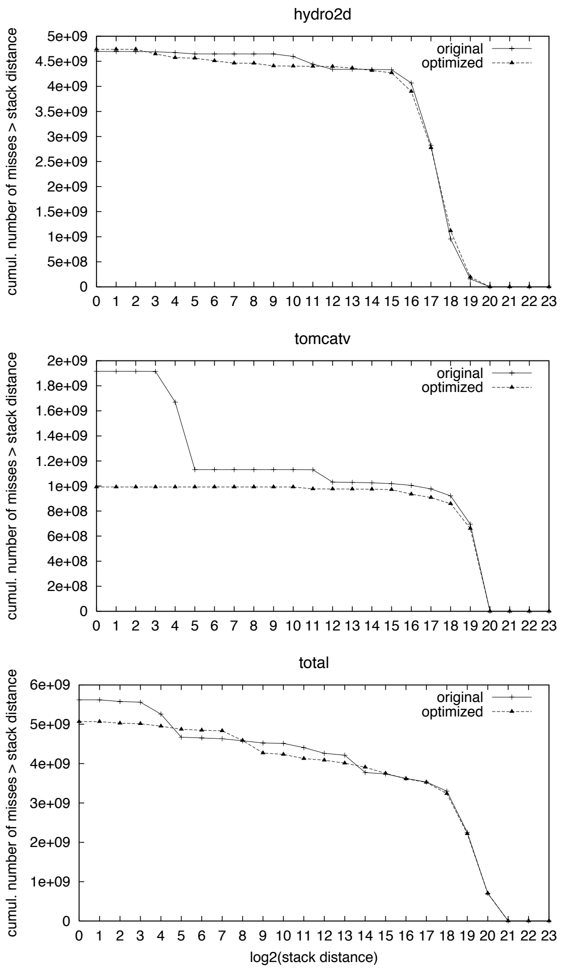

Figure 4. The number of misses with a stack distance exceeding the axis value plotted, both for the original and optimized version. To save space, only the results for `tomcatv` and `hydro2d` are plotted, which represent the two extremes in cache behavior found in the spec95fp benchmarks. The `total` plot shows the stack distance distribution for all programs.

as array padding, loop fusion, distribution, tiling, permutation, data shackling and others. The programs in the SPEC-benchmark were optimized with the appropriate cache parameters. The resulting programs were instrumented to get the stack distance distribution of the cache misses.

In figure 4, the number of cache misses exceeding a given stack distance is plotted for both the original and optimized versions of the programs. This means that the total number of misses can be read at stack distance 0, the number of capacity misses can be read at stack distance 2^{10}, since there are 2^{10} cache lines.

To save space, only the results for `tomcatv`, `hydro2d` and the overall total are plotted. `tomcatv` and `hydro2d` were chosen, since they represent the two extremes of cache behavior in the selected set of programs. `tomcatv` has a substantial number of conflict misses, which are effectively eliminated by the compiler, by applying, amongst others, array padding. `hydro2d` has only a limited number of conflict misses, which can be seen by the small difference of misses with a stack distance greater than 2^0 en 2^{10}.

In the plot for the total of all programs, one can see that the largest number of cache misses occurs at stack distances between 2^{17} and 2^{20}, as is visible by the eye-catching drop in the plot. This means that most misses occur when the same data is referenced, after 2^{17} to 2^{20} other references.

Because the miss rates of the programs differ from 0.6% to 16%, a weighted average is calculated to depict the overall reduction of conflict and capacity misses. The weights in the average for conflict (resp. capacity) misses is chosen to be the ratio of conflict (resp. capacity) misses to total number of references. On average, 30% of the conflict misses are eliminated, whereas only 1.2% of the capacity misses are removed.

The plot in fig. 4 also illustrates that conflict misses are reduced, visualized by the gap between the lines for stack distances smaller than 2^{10}. Capacity misses are not reduced a lot since the gap decreases for stack distances larger than the cache size.

4. Capacity Miss Reduction

The measurements in the previous section show that capacity misses are the major source of cache misses. In the following subsections, it will be discussed how capacity misses can be reduced at different levels.

4.1 Compiler Level

4.1.1 Loop Fusion

The optimizations in the compiler are not only targeted towards decreasing the number of references with a large stack distance. The high-level optimizations also try to

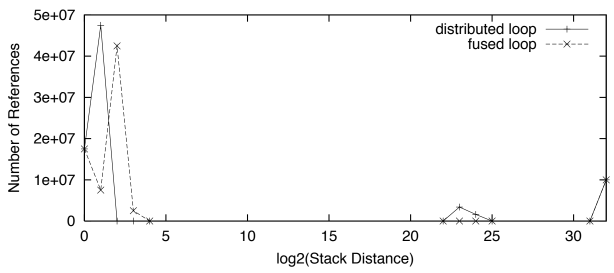

Figure 6. The number of references at a given stack distance is plotted for both the fused and the distributed version of the loops in figure 5, with size parameter `N`=1E7. The references with stack distance larger than 2^{23} will result in cache misses, in a cache with less than 2^{23} cache lines.

parallelize loops so that the amount of exploitable instruction level parallelism increases. Program transformations which create parallel loops can increase the stack distance of references. An example is given in figure 5, where a loop is distributed to create parallel loops. As a side effect, the loop distribution causes the stack distance for the second reference to `A(N)` to increase from 3 to 4`N`-5. For large `N`, this reference will have a stack distance larger than the number of cache lines, and it becomes a cache capacity miss. Therefore, there is a trade-off between reducing capacity misses and increasing the loop parallelism in the program.

The stack distance distribution of the fused and distributed version of the loops in figure 5 are plotted in figure 6. The second reference to `A(I)` in the distributed version occurs at stack distance 2^{23}. In the fused version, all references with reuse occur at a stack distance smaller than 2^4. Observe that in this loop kernel, there are also a substantial number of cold misses, visible as references with a stack distance larger than 2^{31}.

4.1.2 Loop Tiling

Loop tiling is a program transformation aimed at keeping the amount of data referenced between the use and the reuse of the same data smaller than the cache size[7, 6, 1]. In this way, the data is maintained in the cache for a longer processing time, without being refetched from the main memory. An example of this transformation is shown in fig. 7.

This can further be illustrated by applying loop tiling to the matrix multiplication. In figure 8, one can see the stack distance distribution, both for the original and tiled matrix multiplication. This plot illustrates that tiling reduces cache misses by shortening the stack distance of references.

```
DO I=1,N
  A(I) = B(I)
  B(I) = B(I-1)
  C(I) = B(I)
  D(I) = A(I)
ENDDO
```

(a) Fused sequential loop nest

Loop fusion $\longleftarrow$

$\longrightarrow$ Loop distribution

```
DOALL I = 2,N
  A(I)=B(I)
ENDDO
DO I = 2,N
  B(I)=B(I-1)
  C(I)=B(I)
ENDDO
DOALL I = 2,N
  D(I)=A(I)
ENDDO
```

(b) Distributed and parallelized loop nest

Figure 5. An example where loop distribution enables parallel execution of loops, but increases the stack distance for a number of array references. The stack distance d of the second reference to `A(I)` is 3 in the original loop nest, and it becomes $d = 4\mathtt{N} - 5$ in the transformed loop nest.

```
   DO 10 I=1,N
    DO 10 J=1,N
     DO 10 K=1,N
10      A(I,J) = A(I,J) + B(I,K)*C(K,J)
```

(a) Original loop nest

Loop Tiling $\longrightarrow$

```
   DO 10 II=1,N,S1
    DO 10 JJ=1,N,S2
     DO 10 KK=1,N,S3
      DO 10 I=II,MIN(II+S1,N)
       DO 10 J=JJ,MIN(JJ+S2,N)
        DO 10 K=KK,MIN(KK+S3,N)
10    A(I,J) = A(I,J) + B(I,K)*C(K,J)
```

(b) Tiled loop nest

Figure 7. The original and the tiled version of the matrix multiplication. In the tiled version, the iteration space is traversed in blocks. As a result, also the data space is traversed in blocks and long stack distances are reduced.

4.2 Hardware Level

At the hardware level, capacity misses can only be reduced by increasing the capacity of the cache. When the cache size is increased, the stack distance at which cache misses become capacity misses increases. After increasing the cache size, the references which had a stack distance larger than the cache size have a higher probability of hitting in the cache, when the cache size has become larger than their stack distance.

4.3 Algorithmic Level

The fact that a mature compiler was able to remove a considerable portion of the conflict misses, and only a tiny part of the capacity misses indicates that capacity misses are more difficult to eliminate at the compiler level. At the hardware level, the only solution to reducing capacity misses is increasing the cache size, which makes the cache access slower. At the algorithmic level, capacity misses can be reduced by choosing algorithm which generate smaller reuse distances. This shows that capacity misses are more easily reduced at higher levels of application overview. However, optimization at the algorithmic level places an extra burden on the programmer. It is clear that compiler analysis and optimization needs to be extended to remove capacity misses. Furthermore, programming tools must be give feedback to the programmer and the algorithm designer about the long stack distances in their codes. In this way, they know what to focus on when reducing capacity misses at the algorithmic level.

5. Conclusion

The stack distance has been proposed as a metric to measure data locality in programs. With an advanced implementation, it can be measured efficiently, and it is feasible to measure stack distance distributions for complete programs. It predicts the cache behavior of fully-associative LRU caches exactly, since all references with a stack distance smaller than the cache size are hits, while the others are misses. It has been shown that it is also a good predictor for cache miss rates with low set-associativity such as a direct mapped cache.

The stack distance was measured for the SPECfp benchmarks, and it was found that about 70% of the misses are capacity misses, for a 32 KB direct mapped cache. Although the capacity misses dominate the conflict misses,

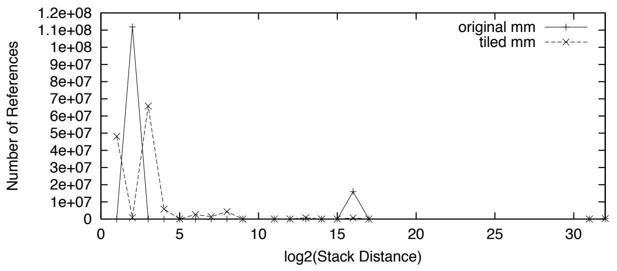

Figure 8. The percentage of references at a given stack distance, both for the original and the tiled matrix multiplication. The matrix size is 400×400. In the tiled version, the references which were originally at stack distance 2^{16} have moved to the left. Because the peak at the large stack distance has been eliminated, most of the cache misses are eliminated.

the majority of traditional cache optimizations focus on conflict misses. The most important compiler optimizations to reduce capacity misses are loop fusion and loop tiling. Implementing hardware techniques to resolve capacity misses efficiently is rather difficult, since a large view on the long data access patterns is needed. The existing optimizations allow to reduce conflict misses by 30% in the SPEC95 benchmark, however the capacity misses are reduced by only 1.2%.

By its nature, the reduction of capacity misses requires a global approach, targeted at a data flow analysis of the program. Possible perspectives are the detection of independent parts, improving locality by mapping communicating components close together and minimizing the average stack distance as a common objective. Future work will address these objectives, keeping a balance with existing parallelizing transformations.

References

[1] K. Beyls and E. D'Hollander. Compiler generated multithreading to alleviate memory latency. *Journal of Universal Computer Science, special issue on Multithreaded Processors and Chip-Multiprocessors*, 6(10):968–993, oct 2000.

[2] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice Hall, Englewood Cliffs, 1973.

[3] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN'95: conference on programming language design and implementation*, pages 279–290, June 1995.

[4] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[5] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, pages 364–373, May 1990.

[6] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *PLDI*, pages 346–357, 1997.

[7] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[8] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999.

[9] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE transactions on computers*, 48(2):142–149, Feb 1999.

[10] C. Pyo and G. Lee. Reference distance as a metric for data locality. In *HPC-ASIA 97*, pages 151–156, 1997.

[11] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, June 1998.

[12] M. S. Schlansker and B. R. R. Cover. EPIC: Explicitly parallel instruction computing. *IEEE Computer*, 33(2):37–45, Feb. 2000.

[13] A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE '93 – Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 305–316, Munich, Germany, June 14–17, 1993. Springer-Verlag.

[14] Sgi pro64 compiler. http://oss.sgi.com/projects/Pro64.

[15] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings, Supercomputing '93*, pages 410–419, March 1993.

[16] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, Chau-Wen, Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994.