Cache Behavior Analysis Without Profiling

Kristof Beyls and Erik D'Hollander

Electronics and Information Systems
Ghent University,
Sint-Pietersnieuwstraat 41,
9000 Gent, Belgium
kristof.beyls@elis.rug.ac.be

Abstract

The growing gap between processor and main memory speed makes it necessary to exploit the caches maximally in order to obtain reasonable program execution speed. Many program transformations have been proposed in order to make the cache behavior better. However, if one wants maximum effectivity from such optimizations, the cache behavior needs to be determined first. In contrast to profile-driven measurement of the cache behavior, this paper presents a method which derives it from the structure of the loops in the program. The lack of profiling makes the time needed to calculate the cache behavior independent of the programs input data. Furthermore, in contrast to other techniques which calculate cache behavior at compiler time, the presented technique is exact and is able to handle fully associative caches efficiently. The efficiency originates from the use of an intermediate data locality metric: the reuse distance. From the reuse distance, the hit/miss behavior of a memory access can be easily determined. Furthermore, the use of this analysis in a cache optimization phase in an EPIC-compiler for the Itanium processor is shown as an example of its practicality.

1 Introduction

For current processors, reaching peak performance for real programs becomes ever harder, mainly because of two reasons. First, the length of the pipeline in the processor is increasing and the number of parallel instructions that can be issued at every clock tick is also increasing. This leads to ever more instructions which are concurrently executing in a processor. To keep the pipelines in the many functional units busy, the processor must be able to predict what path the program is going to follow, tens of instructions in advance. In order to enable this, branch predication hardware is added to the processor. So, a first major cause of underperforming processors is branch misprediction.

The second main cause of underachieving processors originates in the memory wall, the speed difference between processor and memory. Processor execution speed follows Moore's law, i.e. every 18 months, instruction processing capabilities double. Unfortunately, the speed of the memory increases at only about 7% per year. Consequently, the memory wall doubles about every two years. Nowadays, processors can execute up to a thousand instructions during one fetch from main memory. To bridge the speed gap, a number of fast intermediate data caches are located between processor and memory. The different caches each have a different trade-off between size and speed, with the L1-cache being the smallest and fastest. Clearly, if a program exhibits many cache misses, the processor will be data-starved, and will be slowed down a lot. As a rule of thumb, an average program is executing useful instructions about one third of the time, is wasting time because of branch mispredictions a third of the time, and is waiting for data to return from the memory one third of the execution time. In this paper, we focus on reducing the cache miss bottleneck, so that the execution time can be improved.

1.1 Cache Behavior Optimization

The cache behavior of a program can be optimized at three different levels: the hardware level, the compiler level and the algorithm level. At the hardware level, the characteristics of the cache – such as size, associativity and replacement policy – can be chosen, so that for 'the average program', the number of cache misses is minimal. However, in a general purpose machine, it is nearly impossible to optimize the cache for all possible programs. At the compiler level, the program which is compiled is optimized in or-

der to reduce the number of cache misses. If the compiler is not able to minimize the cache misses, the programmer needs to do it itself, possibly by changing the algorithms. Clearly, it is desirable that the compiler can optimize a programs cache behavior maximally, without programmer intervention. A large number of program transformations have been proposed to reduce cache misses[9, 8, 13, 15, 1, 2]. However, before the compiler can decide which transformations are profitable, it needs to know the cache behavior of the program region it would apply them to. In this paper, a method is devised which calculates the cache behavior of sequences of loop regions in the program exactly.

1.2 Cache Bottleneck Identification

The most traditional way to measure a programs cache behavior is profiling[14]. The program is instrumented so that during execution, each access to the memory is traced and fed into a cache simulator. This method gives the exact results, but it's very slow, since each individual memory access needs to be processed. Furthermore, the cache statistics are only measured for a single input to the program. If another input would be used, a different cache behavior could be observed. So, for all relevant inputs, the cache simulation should be repeated.

The limitations of cache simulation has lead researches to seek for analytical ways to extract cache behavior directly from a programs source code, without needing to execute it. One of the earliest steps was taken by Ghosh[6], which developed Cache Miss Equations. However, these cache miss equations were based on reuse vectors, which do not represent the program exactly. As a consequence, the calculated cache behavior is only an approximation of the real cache behavior. Furthermore, the equations cannot easily be solved for high-associative caches. Harper[7] reported another method which was inexact, but it is also efficient for higher associative caches. Chatterjee[4] presented an exact method, based on Presburger arithmetic. However, the method is only efficient for low-associative caches. In contrast to the previous work in this area, this paper aims at an analytical method which is able to calculate the cache behavior both exact and for highly associative caches.

The rest of the paper is organized as follows. In section 2, the reuse distance is introduced. The reuse distance is an intermediate metric for cache behavior and is the key to the efficient and exact calculation of highly-associative cache behavior. In section 3, the program model and the cache behavior calculation

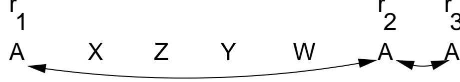

Figure 1: A memory access stream with indication of the reuses. A, W, X, Y and Z indicate the accessed memory location. The accesses to X, Z, Y and W are not part of a reuse pair, since W, X, Y and Z are accessed only once in the stream. $\mathsf{liveset}\langle r_1, r_2 \rangle = \{W, X, Y, Z\}$, and $\mathsf{RD}(\langle r_1, r_2 \rangle) = 4$. $\mathsf{RD}(\langle r_2, r_3 \rangle) = 0$. $\mathsf{FRD}(r_1) = 4$, $\mathsf{BRD}(r_1) = \infty$. $\mathsf{FRD}(r_2) = 0$, $\mathsf{BRD}(r_2) = 4$.

algorithm is described. Some experiments with the algorithm are elaborated in section 4, including application of the algorithm in a cache optimization phase in a EPIC-compiler for the Itanium-processor.

2 Reuse Distance

The reuse distance is defined within the framework of the following definitions.

Definition 1. *A* **reuse pair** $\langle r_1, r_2 \rangle$ *is a pair of memory accesses in a memory access stream, which touch the same memory location, without intermediate accesses to that location. The* **liveset** *of a reuse pair is the set of other memory locations accessed between* r_1 *and* r_2*, and is denoted by* $\mathsf{liveset}\langle r_1, r_2 \rangle$.

Definition 2. *The* **reuse distance** *of a reuse pair* $\langle r_1, r_2 \rangle$ *is the number of unique memory locations accessed between references* r_1 *and* r_2*. It is denoted by* $\mathsf{RD}(\langle r_1, r_2 \rangle)$*, and equals* $|\mathsf{liveset}\langle r_1, r_2 \rangle|$.

Definition 3. *Consider the reuse pairs* $\langle r_1, r_2 \rangle$ *and* $\langle r_2, r_3 \rangle$*. The* **forward reuse distance** *of a memory access* r_2 *is the reuse distance of the pair* $\langle r_2, r_3 \rangle$*. If there is no such reuse pair, its forward reuse distance is* ∞*. The* **backward reuse distance** *of* r_2 *is the reuse distance of* $\langle r_1, r_2 \rangle$*. If there is no such pair, the backward reuse distance is* ∞*. The forward reuse distance of* r_2 *is denoted by* $\mathsf{FRD}(r_2)$*, its backward reuse distance is denoted by* $\mathsf{BRD}(r_2)$.

Example 1. *Figure 1 shows two reuse pairs in a short access stream.*

Lemma 1. *In a fully associative LRU cache with* n *lines, an access with backward reuse distance* $d < n$ *will hit. An access with backward reuse distance* $d \geq n$ *will miss.*

As a consequence of the above lemma, the reuse distance can be used to perfectly predict the cache behavior for fully-associative caches. Furthermore, in [3], it has been shown that the reuse distance can also be used to accurately predict the cache behavior for less associative caches.

3 Cache Behavior Modeling

In the quest for exact and efficient calculation of cache behavior, a tradeoff must be made between the number of programs which fit into the model, and the efficiency with which the cache behavior can be analyzed. If the program model is to simple, only a small number of programs would fit into it, and only for those programs, the cache behavior could be calculated. If the program model would allow to represent overly complex programs, analyzing the cache behavior would become overly inefficient and maybe even intractable without profiling.
In this work, we have chosen to restrict the program to those that can be represented with Presburger formulas. Presburger formulas are those which consists of $\forall, \exists, \vee, \wedge, \neg$ and linear equalities and inequalities over integer variables. An example of such a formula is $\forall x, \exists y : 3x + y = 2 \wedge 1 < x < y \leq N$. This tradeoff allows for a large class of programs to represented (see section 3.1). Furthermore, it is possible to calculate the reuse distance for these programs efficiently, by means of existing tools to simplify the formulas[12] and count the number of solutions[5].

3.1 Program Model

A number of definitions needs to be stated here, in order to be able to concisely and precisely define a program in our cache reuse distance calculation algorithm.

Definition 4. *A* **reference** *is a read or write instruction in the program text. A* **memory access** *is a particular execution of a reference at run time.*

A reference can be executed multiple times in a program, due to loops. For every reference, an iteration space can be constructed, which consists of the loop iterations for which the reference is executed. An example program and a corresponding iteration space is shown in fig. 2. Notice that the dimension of the iteration space equals the number of loop surrounding the reference.

```
DO I=1,N
 DO J=1,I-2
  A(I,J) = A(J,J)
 ENDDO
 B(I)=A(I,1)
ENDDO
```

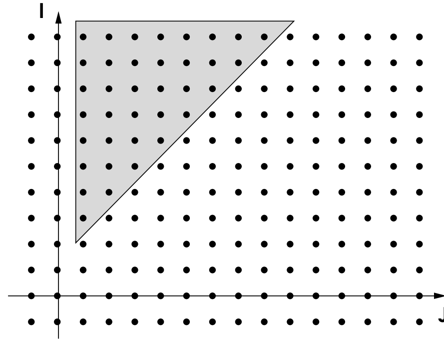

Figure 2: An example program and the iteration space for reference `A(I,J)`.

In order to be able to concisely describe a program below, a number of notations are introduced:

- The set of references in a program is denoted by $\mathcal{R}$. The set of references for the program in fig. 2 is $\{$`A(I,J)`,`A(J,J)`,`B(I)`,`A(I,1)`$\}$.
- For every reference $r \in \mathcal{R}$, the following operators are defined:
  - The iteration space of r is denoted by $[r]$.
    e.g. $[\texttt{A}(\texttt{I},\texttt{J})] = \{(I,J) : 1 \leq I \leq N \wedge 1 \leq J \leq I-2\}$
    $[\texttt{B}(\texttt{I})] = \{(I) : 1 \leq I \leq N\}$
  - The memory element accessed by r at iteration point i is denoted by $r@i$.
    e.g. $\texttt{A}(\texttt{I}-2,\texttt{J})@(4,5) = \texttt{A}(2,5)$.
- The fact that iteration point i is executed before iteration point j is denoted by $i \prec j$.

3.2 Reuse Distance Calculation

The calculation of the reuse distance and the cache behavior of the program occurs in 4 steps:

1. The reuse pairs are calculated.
2. For each reuse pair, the liveset of that pair is calculated.

3. The memory locations in the liveset, as calculated in the previous step, are counted, which leads to the reuse distance.

4. The number of reuse pairs for which the reuse distance is larger than the cache size is counted. This gives the number of cache misses.

The 4 steps are discussed in detail below.

3.2.1 Reuse Pair Calculation

A reuse pair consists of two memory accesses. Each memory access is defined $r@i$, where r is a reference and i is an iteration point. The number of reuse pairs in a memory stream for a program can be huge. In order not to have to enumerate them all separately, we compute the sets of all reuse pairs between two references. Once the references of the first and the second access of the reuse pair are fixed, only the iteration points of the two memory accesses needs to be determined. Therefore, all the reuse pairs for which the first memory access is generated by reference r and the second memory access is generated by reference s is denoted by $\mathsf{reuse}\,(r \longrightarrow s)$. More formally, this set is defined as follows:

$$\forall r, s \in \mathcal{R} : \mathsf{reuse}\,(r \longrightarrow s) = \quad (1a)$$

$$\big\{(I, J) : I \in [r] \wedge J \in [s] \wedge \quad (1b)$$

$$I \lessdot J \wedge \quad (1c)$$

$$r@I = s@J \wedge \quad (1d)$$

$$\forall t \in \mathcal{R} : \neg\,(\exists K \in [t] : I \lessdot K \lessdot J \wedge t@K = r@I)\,\big\} \quad (1e)$$

The constraints above make sure that $\forall (I, J) \in \mathsf{reuse}\,(r \longrightarrow s)$, $\langle r@I, s@J\rangle$ is a reuse pair. Equation (1b) states that I and J must be part of the iteration space of resp. r and s. (1c) says that I must be executed before J; (1d) encodes that the same memory location must be accessed; and (1e) ensures that no intervening memory access touches the same memory location. An example of a code fragment and the corresponding reuse pairs is given in figure 3.

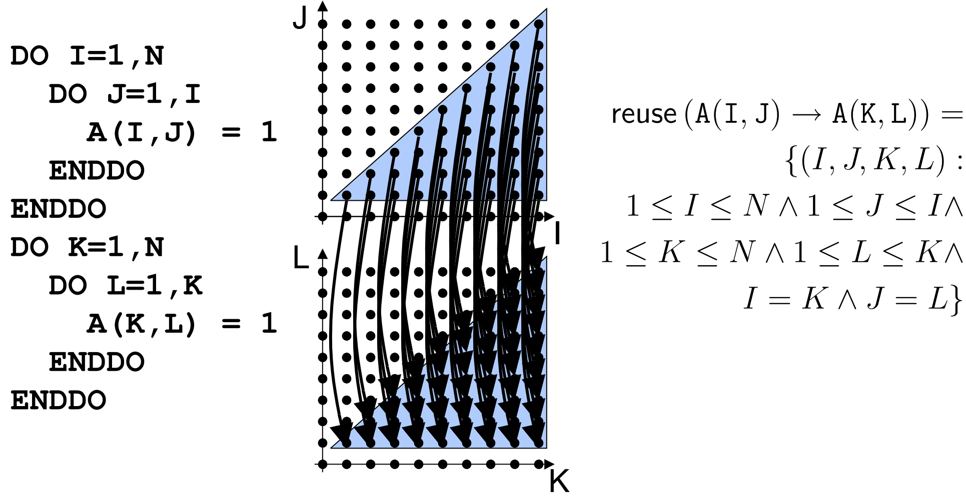

Figure 3: Example code fragment, together with the reuse pairs. On the left side, a code fragment is shown with two references. In the middle, the triangular iteration spaces of both references is shown. For every reuse pair, an arrow is drawn between the iteration points where the reuse occurs. On the right hand side, the mathematical notation of all reuse pairs between `A(I,J)` and `A(K,L)` is shown. It corresponds to the reuse pairs which are shown graphically in the middle.

Furthermore, the following formulas define the iteration points at which forward respectively backward reuse occurs:

$$\begin{aligned}
&\mathsf{reuse_F}\left(r\right) = \\
&\{I_r : \exists s \in \mathcal{R}, J_s \in [s] : (I_r, J_s) \in \mathsf{reuse}\left(r \rightarrow s\right)\} \\
&\mathsf{reuse_B}\left(s\right) = \\
&\{J_s : \exists r \in \mathcal{R}, I_r \in [r] : (I_r, J_s) \in \mathsf{reuse}\left(r \rightarrow s\right)\}
\end{aligned} \tag{2}$$

3.2.2 Liveset Calculation

The second step in the algorithm is to calculate the liveset of the reuse pairs which are found in the first step. For this, 2 auxiliary functions are defined. For every reference r, the data accessed at a given set of iterations is determined by the function map_V. V is the array accessed by r:

$$\mathsf{map}_V = \{I \rightarrow D : I \in [r] \wedge \mathrel{\underset{\cdot}{=}} r@I\} \tag{3}$$

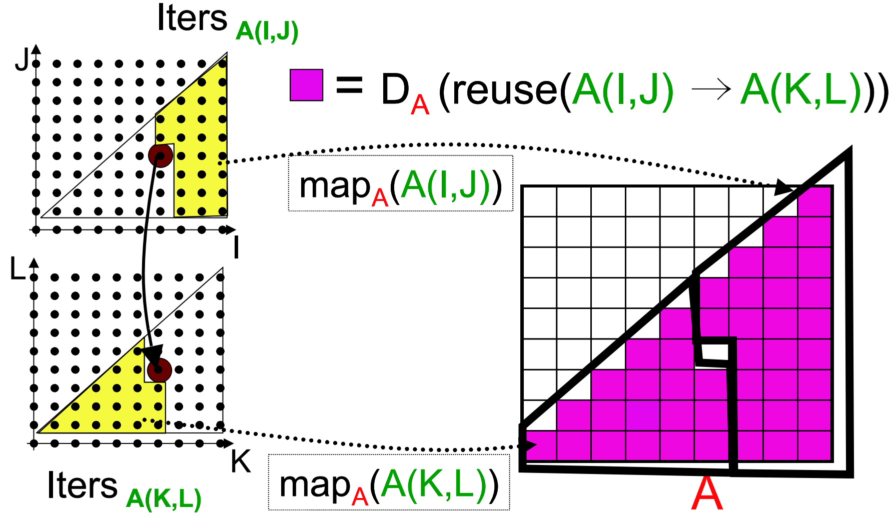

Figure 4: Example of the calculation of the liveset of a single reuse pair. On the left hand side, the iterations that are executed between use and reuse, which are computed by the iters() function, are indicated. These iterations are the input to the map function, which maps iterations to the data they access. The liveset of the reuse pair is indicate on the right hand side.

The second auxiliary function is $\mathsf{iters}_t((I, J))$, which represents the iterations of reference t that are executed between iteration I and J:

$$\mathsf{iters}_t((I, J)) = \{K : I \prec K \prec J\} \tag{4}$$

Given the above functions, it is easy to find all the data (=the liveset) that is accessed between use and reuse at particular iteration points. $\mathsf{D}_V\left(\mathsf{reuse}\left(r \rightarrow s\right)\right)$ represents the data that is accessed between reuses from reference r to reference s:

$$\mathsf{D}_V\left(\mathsf{reuse}\left(r \rightarrow s\right)\right) = \bigcup_{t \in \mathcal{R}} \mathsf{map}_V(t) \circ \mathsf{iters}_t(\mathsf{reuse}\left(r \rightarrow s\right)) \tag{5}$$

An example of the application of the above functions is given in figure 4.

3.2.3 Liveset Counting

The reuse distance of a reuse pair is the amount of data that is accessed between use and reuse. It is simply the number of unique memory locations

in the liveset. Therefore, the reuse distance $\mathsf{RD}(\mathsf{reuse}\,(r \rightarrow s))$ of reuse pairs $\mathsf{reuse}\,(r \rightarrow s)$ is:

$$\sum_{V \in \mathcal{V}} |\mathsf{D}_V\,(\mathsf{reuse}\,(r \rightarrow s))| \tag{6}$$

Different methods for counting the number of solutions of a Presburger formula have been described in detail in [11] and [5].
In the example in figure 4, assuming that the triangle is $N \times N$, the number of data elements accessed is $\frac{N^2+N)}{2} - 1$.

3.2.4 Cache Miss Counting

The last step in the algorithm is to use lemma 1 to compute the number of cache misses in the program. According to the lemma, at every iteration point where the reuse distance is larger than the cache size CS, a cache miss occurs. On can even easily make the distinction between capacity and cold misses: the capacity misses are those for which backward reuse occurs, and the reuse distance is larger than the cache size. The cold misses occur at the iteration points where no backward reuse occurs:

$$\begin{aligned} \mathsf{CAPM}(r) &= \{I : \mathsf{BRD}(()r) \geq CS \wedge I \in \mathsf{reuse_B}\,(r)\} \\ \mathsf{COLDM}(r) &= \{I : I \in [r] \wedge I \notin \mathsf{reuse_B}\,(r)\} \end{aligned} \tag{7}$$

As an example, the same code as in figures 3 and 4 is considered here:

$$\begin{aligned} \mathsf{CAPM}(\mathtt{A}(\mathtt{K},\mathtt{L})) &= \{(K,L) : \mathsf{BRD}(()r) \geq CS \wedge (K,L) \in \mathsf{reuse_B}\,(r)\} \\ &= \{(K,L) : \frac{N^2+N}{2} - 1 \geq CS \wedge 1 \leq K \leq L \leq N\} \\ \mathsf{COLDM}(\mathtt{A}(\mathtt{K},\mathtt{L})) &= \{\} \\ |\mathsf{CAPM}(\mathtt{A}(\mathtt{K},\mathtt{L}))| &= \begin{cases} \frac{N^2+N}{2} & \text{if } \frac{N^2+N}{2} - 1 \geq CS \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

4 Experiments

The generation of the equations was implemented in the FPT-compiler[16]. The Omega-library[12] was used to simplify the equations, the Polylib-library[10] was used to count the number of solutions of the equations. A number of loop nests with different code structures where tested. The results from the analytical calculation where identical to cache simulation in all cases.

Furthermore, it was tested and found that the equations could be generated for most loop nests in the programs from the SPECfp benchmark. However, some of the generated equations took exponential time for the simplification and counting. In the future, we will look into this and find out what equation patterns generate exponential execution time.

Application: Cache Hints

As an example of where the analytical method could be applied in a compiler optimization phase, we show how it can be used for selecting appropriate cache hints. Cache hints indicate at which cache level the data should be expected. Traditionally, compilers assume that all data comes from the L1-cache. With cache hints, the compiler is better informed about the true latency of the load and can try to execute parallel instructions between the start of the load and when the data is returned from the memory. Previously cache hints were selected based on profiling[2]. As an example, we analyze the matrix multiplication:

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

The analytical method described above calculates following backward reuse distances for the different references:

reference	C(I,J)	A(K,I)	B(K,J)
backward reuse distance	0	$2N+1$	N^2+3N

Based on the above reuse distances, and lemma 1, the compiler can calculate for which values of the parameter N, from which cache level the data arrives. Here, it is assumed that there are 2 cache levels, the first cache level can contain 12K data elements, and the second level can retain 256K data elements, as is the case for floating point data on the Itanium processor. From this, the cache level where the data can be found is generated, which is shown in table 1.
The execution time of the analysis took 1 second. In comparison, profiling took 30 minutes, for $N = 400$. For $N = 800$ it took 10 hours. This clearly shows a speed advantage over profiling. Furthermore, with profiling the

N	`C(I,J)`	`A(K,I)`	`B(K,J)`
$1 \ldots 109$	L1	L1	L1
$110 \ldots 510$	L1	L1	L2
$511 \ldots 6143$	L1	L1	Mem
$6144 \ldots 128K - 1$	L1	L2	Mem
$128K \ldots$	L1	Mem	Mem

Table 1: The cache level where the data is found, for the matrix multiplication, for different references and different matrix sizes N.

border values of N for different cache behavior cannot easily be determined with one measurement, whereas in the analytical method, the cache behavior is easily calculated for all values of N.
After the appropriate cache hints were inserted for this matrix multiplication, the stall time because of cache misses dropped from 10% to below 1% on an 733Mhz Itanium-processor.

5 Conclusion

In this paper, a mathematical formulation of cache behavior was presented. The method is based on Presburger formulas and counting their number of solutions. The correctness was checked by comparing it with cache simulation, where the results were always identical. The experimental results further show that the analytical method has a number of advantages of profiling. It is much quicker than profiling and is independent of the execution time of the program. Furthermore it allows to parameterize the program, so that the cache behavior is calculated for different data sizes simultaneously. The generation of the formulas is easy. However, some program constructs lead to exponential computation time for counting the number of solutions. In the future, we will analyze which program constructs cause this and look for alternative ways to represent those constructs. Furthermore, the model will be extended to larger cache line sizes and set associativity. Also, applicability in a compiler will be further extended for different cache optimizing transformations, such as tile size selection, loop fusion, cache remapping, etc.

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] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Euro-Par*, 2002.

[3] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, 2001.

[4] S. Chatterjee, E. Parker, P. Hanlon, and A.R.Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI*, 2001.

[5] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, May 1996.

[6] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour*. PhD thesis, Princeton University, November 1999.

[7] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transaction on Computers*, 48(10):1009–1024, oct 1999.

[8] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[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] The polyhedral library. http://icps.u-strasbg.fr/PolyLib/.

[11] W. Pugh. Counting solutions to Presburger formulas: How and why. *ACM SIGPLAN Notices*, 29(6):121–134, jun 1994.

[12] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM TOPLAS*, 20(3):635–678, May 1998.

[13] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In V.N.Alexandrov, J. Dongarra, and C.J.K.Tan, editors, *Proceedings of the 2001 International Conference on Computational Science*, Lecture Notes in Computer Science, San Francisco, CA, USA, May 28-30, 2001. Springer-Verlag.

[14] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[15] Y. Yan, X. Zhang, and Z. Zhang. Cacheminer: A runtime approach to exploit cache locality on SMP. *IEEE Transactions on Parallel and Distributed Systems*, 11(4):357–374, Apr. 2000.

[16] F. Zhang. *The FPT Parallel Programming Environment*. PhD thesis, Ghent Unversity, 1996.