Estimating Cache Performance for Sequential and Data Parallel Programs

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, AUSTRIA
e-mail: tf@par.univie.ac.at

Published in Proc. of HPCN'97, Springer Lecture Notes in Computer Science, Vienna, Austria, April, 1997

Abstract

This paper introduces an analytical model that enables automatic estimation of the cache performance for both sequential and data parallel Fortran programs. The estimation is based on a classification of array accesses with respect to cache reuse at the source code level. An estimated upper bound of the number of distinct cache lines accessed inside of a loop is statically computed. Based on this estimate the number of cache misses for loops, procedures and the entire program can be predicted.

The method has been implemented as part of P^3T (Parameter based Performance Prediction Tool) and successfully supports *VFCS* (Vienna Fortran Compilation System) in guiding the application of data distributions and program transformations on distributed memory multiprocessor systems to achieve greater cache effectiveness. Experiments are presented that demonstrate the efficacy of our approach with very encouraging experimental results.

1 Introduction

Intense efforts have been invested to increase the computing capabilities for state-of-the-art processors in the last decades. Unfortunately the memory bandwidth has not been improved nearly as good as the processor's speed. Advanced memory strategies to reduce processor idle times induced by memory bottlenecks are required to balance the interplay between processor and memory.

A memory hierarchy usually consists of a set of registers, cache and main memory. Increasing the size of each memory component in terms of bytes is one feature to enhance. However, the data layout into the memory hierarchy still remains a crucial problem. For sequential programs it is well known that inefficient data mapping into the memory hierarchy – *data locality problem* – causes critical performance degradation. Introducing distributed memory multicomputers increases the memory bandwidth but implies a severe bottleneck in case that non-local data must be transferred from remote processors.

It has been shown ([6, 5, 2]) that improving the critical cache performance results in considerable reductions in runtime. Predicting the distinct number of cache lines accessed inside of a nest of loops has seen considerable research activity during the last few years. In general it is assumed that the smaller this number the smaller the critical number of cache misses.

J. Ferrante, V. Sarkar and W. Trash ([4]) compute an upper bound for the number of cache lines accessed in a sequential program, which allows them to successfully guide loop interchange. Part of their techniques are based on polynomial evaluations.

K. Kennedy and K. McKinley ([5]) developed a cost function based on data dependences to evaluate the goodness of data locality across the iterations of the innermost loop only. They do not consider important machine dependent parameters such as cache line size and available cache lines for their model.

M. Wolf and M. Lam ([6]) base their cost model on a reuse vector space in combination with a localized iteration space. This allows them to accurately evaluate the cache behavior of a program at the cost of a higher computational complexity.

Most previous work is restricted to sequential or shared memory parallel programs. The impact of data distributions and optimizing program transformations – such as loop interchange, distribution, tiling, and so forth – on the cache performance of distributed memory multiprocessor systems has not been sufficiently taken into account.

This paper describes an automatic analysis of a program's cache usage to support a compiler or programmer in achieving greater cache effectiveness. An algorithm is presented that estimates the number of cache misses for sequential and data parallel Fortran programs. It is based on analyzing all array references in a program's loops, classifying them regarding data reuse, and computing a cost function for these classes that describes the cache behavior of the program. It is shown how to extend this cost function to procedures and entire programs. For a parallel program, an upper bound of the number of cache misses across all processors is computed.

The most critical factors, such as cache line sizes, cache size, data types, and so forth, are modeled to reflect the performance impact of the target machine. The described analysis techniques are applicable to other cache architectures and memory hierarchy levels as well. Careful attention has been paid to strictly separate machine dependent from machine independent parameters in the cache cost function which alleviates its portability to other architectures.

The rest of the paper is organized as follows: Section 2 formally defines array access classes. The next section describes in detail how to automatically determine a cache cost function for loops, procedures and an entire program. Section 3 presents experimental results on automatic cache performance prediction for the JACOBI relaxation code, that show the importance of using good performance analysis techniques to guide the application of data distributions and program transformations. Finally, Section 4 presents conclusions and ideas for future work.

2 Statically Estimating Cache Performance

In this section a formal definition of array access classes is presented. Furthermore, an algorithm to compute an upper bound for the estimated number of cache misses implied by an array access class, a loop nest, a procedure and an entire program is described.

It is assumed that the loops of a loop nest L are consecutively numbered from 1 to n, where the loop at level 1 is the outermost and at level n the innermost loop. The loop at level i is denoted by L_i. Loop L_i consists of a loop body and a loop header statement. loop increment is equal to 1. L_u^l represents the set of loops between loop nest level l and u $(1 \leq l \leq u \leq n)$. I_i is the loop variable of L_i. I_u^l refers to the set of loop variables of L_u^l.

Only the data cache of the memory hierarchy is modeled. A cache line is the atomic data unit to be transferred between main memory and data cache. The overall number of cache lines available is specified by OCL. The cache line size specified in terms of bytes is denoted by CLS. OCL, CLS and the byte length of different data types are the only architecture specific parameters required for the described cache cost function.

A random replacement strategy is assumed for our cache performance analysis.

2.1 Array Access Classes (ACs)

The definition of an array access class is based on a specific number of k innermost loops of a not necessarily perfectly nested loop. Two array references of the same array are in the same AC with respect to a loop nest L_n^l if they access some common memory location in the same array dimensions and reuse occurs across loop iterations of L_n^l.

Definition 2.1 Array Access Class

Let $\mathcal{R} = \{r_1, ..., r_q\}$ be a set of references inside of a loop nest L_n^l $(1 \leq l \leq n)$, then $\mathcal{R}$ specifies an array access class AC, iff all references in $\mathcal{R}$:

1. *access the same m-dimensional array A, and*
2. *have identical subscript expressions in all array dimensions, or*
   - *in at least $m - k$ $(k = \min(m, n - l + 1))$ equal dimensions $D_e = \{dim_{j_{k+1}}^A, ..., dim_{j_m}^A\}$ the subscript expressions of all references in $\mathcal{R}$ must be identical and may not contain any loop variables of I_n^l, and*

- *there are at most k equal dimensions* $D_v = \{dim_1^A, ..., dim_k^A\}$*, such that for each* $q \in D_v$ *there exists at least one reference in* $\mathcal{R}$ *that contains at least one loop variable of* I_n^l *in the* q-th *dimension subscript expression.*

dim_i^A is a projection to the i-th dimension of an m-dimensional array A. Let $\{j_1, ..., j_k\} \subseteq \{1, ..., m\}$ and all pairs (j_r, j_s) are different with $r \neq s$ $(1 \leq r, s \leq k)$. Therefore $dim_{j_i}^A$ with $1 \leq i \leq k$ describes an arbitrary dimension of A not necessarily the i-th one.

2.2 Loop Cost Function

In the following the number of cache misses for a loop nest L is defined. P defines the set of processors executing L. $\mathcal{F}_L$ is the set of procedure call statements contained in L. $proc(q)$ is the associated procedure of a procedure call $q \in \mathcal{F}_L$. $call(E)$ is the set of all corresponding calls to a procedure E. $cmE(E)$ (see Definition 2.3) is the accumulated number of transfers implied by E. $freq(S)$ specifies the statement execution count for a statement S.

Definition 2.2 Number of Cache Misses for a Loop Nest

The number of cache misses induced by a loop L with respect to a processor $p \in P$ *is defined by a total function* $cmL^p(L)$*.*

$$cmL(L) = \frac{1}{|P|} \sum_{p \in P} cmL^p(L) + \sum_{q \in \mathcal{F}_L} \frac{cmE(proc(q)) * freq(q)}{\sum\limits_{g \in call(proc(q))} freq(g)}$$

defines the overall number of cache misses induced by L with respect to all processors in P.

The first term in Definition 2.2 describes the mean value of cache misses implied by L across all processors in P executing L. The second term is explained as follows: In order to take procedure calls into account, the parameter outcome for a single procedure call instantiation is supposed to be independent of the call site. This means that the parameter outcome at a particular call site is the same as the parameter outcome of the procedure over all call sites, which is a common assumption made for performance estimators. All call graphs are supposed to be acyclic. Note that Definition 2.2 is also applicable to a sequential program iff $|P| = 1$.

In order to calculate $\overline{cmL^p}(L)$, the estimated number of cache misses for a specific processor $p \in P$ with respect to L, an iterative algorithm is incorporated. First, this algorithm computes cl_n, the estimated number of cache lines accessed by p in L_n^n, the innermost loop of L. If cl_n is greater than OCL, then executing all iterations of the innermost loop does not require all cache lines available on the target processors p. The array references appearing in the innermost loop are re-grouped into array access classes on the basis of L_n^{n-1} according to Definition 2.1 for $l = n-1$. This results in a new amount of cache lines cl_{n-1} accessed in the innermost 2 loops of L. The iterative algorithm continues to compute cl_i $(1 \leq q \leq i \leq n)$ for the next higher loop levels until cl_q (L_q is defined as the *cache overflow loop*) is larger than OCL. Then the current cl_q is multiplied by the iteration count product of all loops in L_{q-1}^1. This yields an upper bound for the number of cache lines accessed – which is assumed to correlate with the number cache misses – in L:

$$\overline{cmL^p}(L) = cl_q * \prod_{i=1}^{q-1} iter(L_i) \tag{1}$$

$iter(L_i)$ specifies the average number of loop iterations for L_i as derived by the *Weight Finder* (a profiler for sequential Fortran programs). For a parallel program $iter(L_i)$ must be adapted. *VFCS* applies a loop iteration elimination transformation ([1]) to all loops in a parallel program, that eliminates redundant loop iterations for every specific processor. This is done by changing the loop bounds appropriately: Minimum and maximum functions of the original loop bounds and local array segment boundaries – of the array section owned by a local processor – specify the loop bounds of a parallel program. In order to compute $iter(L_i)$ for a specific processor in a multiprocessor system, the loop bounds of the parallel program are parsed once and evaluated for every processor.

Equation 1 shows an optimistic formula as it assumes that the cache fills up after the first iteration of L_{q-1}. Actually the cache already fills up during the first iteration of L_{q-1}. Therefore, let $r_1 = 1 - (OCL/cl_q)$ which is proportional to the cache lines not fitting into the cache as required by L_n^q. Furthermore, if $cl_{q+1} = 0$ then $r_2 = N_{acs} * iter(L_q) - cl_q$ with N_{acs} is the number of array access classes in L_n^q, otherwise $r_2 = cl_{q+1} * iter(L_q) - cl_q$ which is the estimated difference between a pessimistic and an optimistic number of accessed cache lines for all array references in L_n^q. Consequently a more precise formula for $\overline{cmL^p}(L)$ is:

$$\overline{cmL^p}(L) = (cl_q + r_1 * r_2) * \prod_{i=1}^{q-1} iter(L_i) \tag{2}$$

The term $r_1 * r_2$ compensates for additional cache lines accessed during the first iteration of L_{q-1}, which is ignored in Equation 1.

Furthermore, $\overline{cmL}(L)$ is defined by:

$$\overline{cmL}(L) = \frac{1}{|P|} \sum_{p \in P} \overline{cmL^p}(L) + \sum_{q \in \mathcal{F}_L} \frac{\overline{cmE}(proc(q)) * freq(q)}{\sum\limits_{g \in call(proc(q))} freq(g)} \tag{3}$$

Note that $\overline{cmL}(L)$ – in contrast to $\overline{cmL^p}(L)$ – models also the number of cache misses induced by procedure calls appearing in L.

Assume that there is a set of h_k array access classes for L_n^k. The estimated number of cache lines required for all array references in L_n^{k+1} is defined by:

$$cl_k = \max(cl_{k+1}, \sum_{j=1}^{h_k} cl_k^{ac_j}) \tag{4}$$

where $cl_k^{ac_j}$ is the number of cache lines accessed by array access class ac_j in L_n^k. Array access classes can be merged at a higher loop nest level. This may induce $cl_{k+1} > cl_k$ which is impossible to occur in practice. In order to prevent this anomaly a maximum function (max) is incorporated to guarantee that $cl_k \geq cl_{k+1}$.

cl_k^{ac}, the number of estimated cache lines accessed by all references in ac with respect to L_n^k, is computed as follows:

1. If all references in ac are loop invariant with respect to I_k in all array dimension subscript expressions, then $cl_k^{ac} = cl_{k+1}^{ac}$.

2. If for all array references in ac there exists at least one array containing I_k in the first array dimension subscript expression, and there does not exist an array in ac containing I_k in any dimension subscript expression other than the first one, then cl_k^{ac} $= (cl_{k+1}^{ac} * iter(L_k))/CLS + 1$. In this equation *cache line overlap* is modeled by adding 1 to cl_k^{ac}.

3. otherwise $cl_k^{ac} = cl_{k+1}^{ac} * iter(L_k)$

The references of several array access classes $\{ac_1, \ldots, ac_q\}$ at L_{k+1} can be merged to a new array access class ac_r at L_k, such that $cl_{k+1}^{ac_r} = \max(cl_{k+1}^{ac_1}, \ldots, cl_{k+1}^{ac_q})$. It is important that the maximum – instead of the sum – of the number of cache lines accessed across all array access classes is computed, as the references merged can exploit data reuse.

2.3 Procedure and Program Cost Function

Extending the cache cost function to a procedure or a program is straight forward. The estimated number of cache misses for every specific loop is weighted by its statement execution count ($freq$).

Definition 2.3 Number of Cache Misses for a Procedure or a Program

Let E be a procedure or an entire program, E contains a set of nested loops $\mathcal{L}_E$, $freq(L)$ is the execution count of $L \in \mathcal{L}_E$, and $\mathcal{F}_E$ is the set of procedure calls – outside of loops – in E, then the number of cache misses induced by all loops in $\mathcal{L}_E$ is defined as follows:

$$cmE(E) = \sum_{L \in \mathcal{L}_E} cmL(L) * freq(L) + \sum_{q \in \mathcal{F}_E} \frac{cmE(proc(q)) * freq(q)}{\sum\limits_{g \in call(proc(q))} freq(g)}$$

The first term in Definition 2.3 corresponds to the loops contained in E. Each loop is weighted by its execution count to reflect its impact on the overall program performance. The second term models procedure calls outside of loops in E. It is assumed that the same cache behavior is implied by every instantiation of L.

The estimated value for $cmE(E)$ is denoted by $\overline{cmE}(E)$. This value is computed by incorporating $\overline{cmL}(L)$ instead of $cmL(L)$ according to Equation 3 in Definition 2.3. Note that cmL and cmE are upper bounds for the number of cache misses induced.

3 Experiments

In this section we report on several experiments that show the effectiveness of our cache cost function to rank program versions with respect to their cache performance. The *Weight Finder* is used to derive characteristic values for program unknowns, *VFCS* to process both sequential and parallel programs. The described cache cost function has been implemented as part of P^3T [2, 3] which automatically computes this cost function at compile time. In order to validate our approach, a variety of kernel programs were executed and measured on a iPSC/860 hypercube with 16 processors.

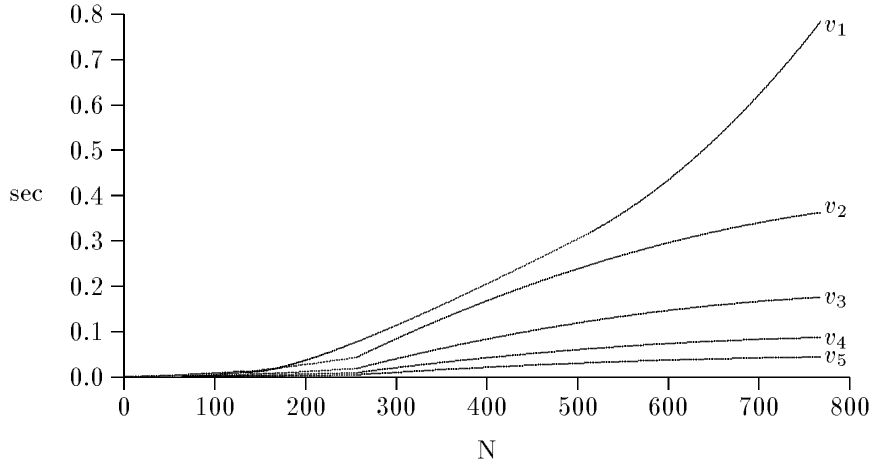

Figure 1: JACOBI runtime for various data sizes and number of processors

The described parameter is used to rank different program versions with respect to their cache behavior. This is validated by executing the programs on the iPSC/860 hypercube, measuring their runtime and comparing them with the predicted cache performance results. Results will demonstrate that there exists a close correlation between our cache cost function and the corresponding measured runtime.

3.1 Jacobi Relaxation

This experiment will demonstrate that our cache cost function is able to correctly rank various versions of a Jacobi relaxation by applying different data distributions and loop interchange for a variety of processor numbers and problem sizes. The performance ranking will be compared against runtime measurements taken on the iPSC/860 hypercube. The following program segment illustrates the Jacobi relaxation kernel:

Example 3.1

```
          DO J=2,N-1
            DO I=2,N-1
S:            Unew(I,J) = (1-Omega)*U(I,J)+Omega * 0.25*(F(I,J) +
     *                     U(I-1,J) + U(I+1,J) + U(I,J+1) + U(I,J-1))
            ENDDO
          ENDDO
```

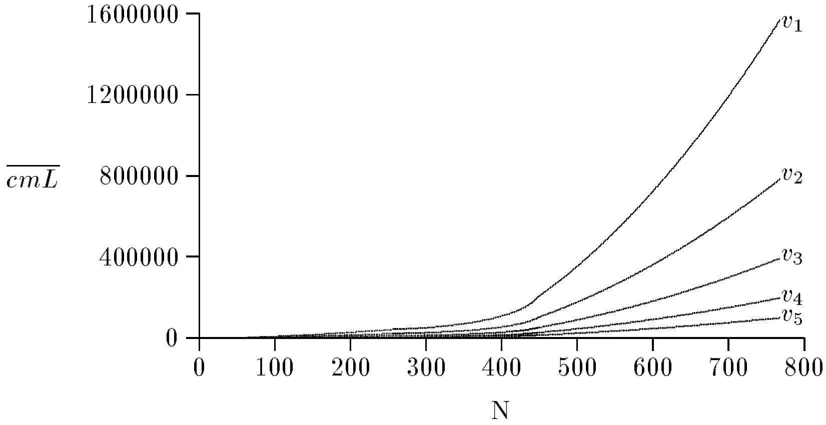

Figure 2: Number of cache misses in JACOBI for various data sizes and number of processors

By using *VFCS* all kernel arrays were column-wise distributed to 1, 2, 4, 8, and 16 processors, which yields 5 program versions $v_1, \ldots, v_5$, respectively. Note that v_1 is a sequential code while all others are parallel versions. The intermediate representation of the parallelized code v_5 – written in HPF style – under *VFCS* is the following:

Example 3.2

```
           PARAMETER (N=1024)
!HPF$      PROCESSORS :: P(16)
           REAL Unew(N,N), U(N,N), F(N,N)
!HPF$      DISTRIBUTE(*,BLOCK) ONTO P :: Unew, U, F
           ...
           EXSR U(:,:) [0/0,1/1]
           EXSR F(:,:) [0/0,1/1]
           mask → DO J=2,N-1
              mask → DO I=2,N-1
S:                 Unew(I,J) = (1-Omega)*U(I,J)+Omega * 0.25*(F(I,J) +
        *             U(I-1,J) + U(I+1,J) + U(I,J+1) + U(I,J-1))
              ENDDO
           ENDDO
```

The HPF *PROCESSOR* directive specifies a one-dimensional processor array P with 16 processors. The HPF *DISTRIBUTE* directive defines a column-wise distribution of arrays Unew, U, and F onto P. The loop header statements are masked by a boolean guard (*mask*) which specifies that a processor p only executes a loop iteration I if the left-hand-side array element $Unew(I)$ in statement S is owned by p. Non-local data, that is required to compute local values for $Unew(I, J)$ in statement S, is exchanged outside the loop nest by EXSR (communication) statements.

We executed $v_1, \ldots, v_5$ and measured their runtimes (see Figure 1) on the iPSC/860 hypercube for various data sizes N (array dimension size). The y-axis specifies the runtime in seconds of the program versions, measuring only the double nested loop without enclosing communication statements. It can be seen, that every doubling of the number of processors incurs almost 50 % reduction in runtime. Therefore, almost linear speedup is achieved by the underlying parallelization system.

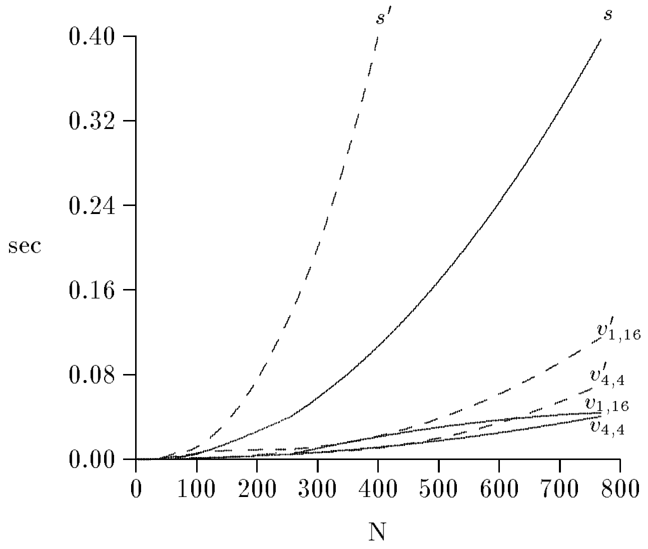

Figure 3: Runtime of the Jacobi kernel before and after loop interchange

Figure 2 illustrates the corresponding estimated number of cache misses ($\overline{cmL}$) implied by every different program version for various data sizes N. The cache cost functions have a very similar shape as the runtimes plotted in Figure 1. It can be seen that each time the number of processors is doubled the number of cache misses is approximately cut in half. Clearly, the cache performance of a parallel program is proportional to its size. For a scenario where the number of cache misses would not decrease by increasing number of processors, then linear speedup – as shown in Figure 1 – most likely could not be achieved. The more processors are involved in the computation, the larger the overall distributed cache size and the better the overall cache behavior. Thus the number of cache misses decreases by increasing the number of processors which is correctly reported by the described cache cost function.

Applying loop interchange [1] to the code in Example 3.1 yields following code:

Example 3.3

```
        DO I=2,N-1
          DO J=2,N-1
S:          Unew(I,J) = (1-Omega)*U(I,J)+Omega * 0.25*(F(I,J) +
     *             U(I-1,J) + U(I+1,J) + U(I,J+1) + U(I,J-1))
          ENDDO
        ENDDO
```

In order to evaluate the cache impact of interchanging the two loops of Example 3.1, we executed and measured the runtime of three program versions before and after loop interchange: sequential program s; program $v_{1,16}$, which is based on column-wise distributing all arrays to 16 processors; and program $v_{4,4}$ based on 2Dblock-wise distributing all arrays to a 4x4 processor array. The corresponding program versions after loop interchange (Example 3.3) are denoted by s', $v'_{1,16}$ and $p'_{4,4}$, respectively. Figure 3 illustrates the measured runtimes for all six program versions by varying the data size N. It can be seen, that interchanging the loops decreases the performance of s, $v_{1,16}$ and $v_{4,4}$.

Figure 4 correctly confirms that interchanging the loops has a detrimental effect on the cache behavior of the program versions. The estimated number of cache misses increases for all three program versions after loop interchange. In Figure 3 plots a measured runtime of $v'_{4,4}$ that is clearly superior to that of $v'_{1,16}$. According to Figure 4, at first hand, the number of estimated cache misses does not correspond to this observation. $\overline{cmL}$ is almost identical with respect to $v'_{4,4}$ and $v'_{1,16}$. To be precise, about 65 % of all $\overline{cmL}$ values for $v'_{4,4}$ are smaller than those of $v'_{1,16}$. Based on this percentage value it can be correctly predicted that the cache behavior of $v'_{4,4}$ is superior to that of $v'_{1,16}$. The reason why our cache cost

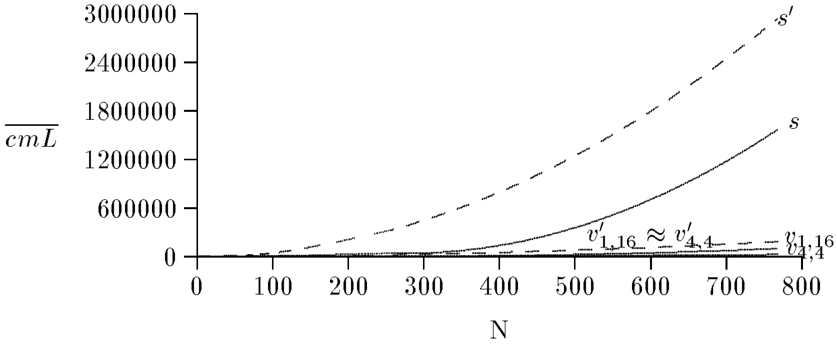

Figure 4: Cache misses of the Jacobi kernel before and after loop interchange

function does not report exactly the same ranking as for their runtimes can be explained as follows:

- The inner loop of the kernel iterates across the second array dimension after loop interchange. If the block-sizes in the second array dimension are large enough for both distribution strategies (2Dblock and column-wise distribution) then the cache fills up in the first iteration of the outer loop. For this case – according to our loop cost function – the number of cache lines accessed in the inner loop is multiplied by the loop iteration count of the outer loop. This yields almost identical values of $\overline{cmL}$ for both program versions. However, if the cache is not full after the first iteration of the outer loop, then the cache performance of $v'_{1,16}$ should be superior if compared with $v'_{4,4}$. Consequently, based on Fortran's column major-order storage policy and the outer loop iterating across the first array dimension, $v_{1,16}$ should induce less cache misses than $v_{4,4}$.

- Equation 3 does not model cache interferences where two references are mapped to the same cache line. Cache interferences can make the cache behave erratically. It may well be the case that cache interferences are the predominating factor for the cache performance of $v'_{1,16}$ but has only little impact on $v'_{4,4}$.

- The disparity in runtime between the two program versions is influenced by the pipelined integer and floating point units of the iPSC/860 processor which again is not modeled by our cache analysis.

Overall we observed that our cache cost function correctly estimated the cache behavior of more than 80 % of all experiments done so far. Our automatic cache performance analysis enables an efficient ranking of different program versions considering both data distributions and program transformations with respect to the actual cache behavior.

4 Conclusion

This paper addresses the problem of automatically estimating the cache behavior of sequential and data parallel programs at compile time. Our approach is based on classifying array accesses inside of nested loops with respect to data locality and reuse. Both temporal and spatial reuse are modeled. While previous work primarily classifies array references incorporating data dependence information, this approach is based on varying array dimensions with respect to loop variables. All array references in nested loops are analyzed. They are partitioned into array access classes based on data reuse and locality. Finally, an upper bound of the associated number of cache misses reflects the cache behavior. It is shown how to extend this cost function to procedures and an entire program. The methodology models data type sizes, number of cache lines available, cache line size, cache overflow loop, cache line overlap, spatial and temporal reuse, merging of array access classes, and so forth, to provide a realistic cache cost function. This function depends on the underlying architecture with respect to following parameters: overall amount of cache lines available, cache line size and data sizes for different data types. A clear separation of machine specific and program dependent parameters supports the portability of the cache cost function to other architectures.

Because we accept some imprecision by applying our analysis at the source code level, our algorithms are less complex than others, and they can be applied to computations that have not been considered by other researchers. For instance, we can support imperfectly nested loops, multiple loop nests, procedures, and entire programs, and do not depend on any data dependence analysis. We believe that this approximation is a very reasonable one, especially in view of the fact that we use our cache performance analysis as part of a state-of-the-art parallelizing compiler which requires strong portability of large portions of its analysis.

Experimental results provide strong evidence that the proposed cache cost function successfully guides the application of different data distribution strategies and program transformations such as loop distribution, fusion and interchange, in order to tune the cache performance. A realistic ranking of different program versions with respect to the actual cache behavior can be made based on this evaluation. The cache cost function has been validated against runtime measurements which were taken on the iPSC/860 hypercube. The cache cost function is implemented as part of the Parameter based Performance Prediction Tool (P^3T), which is integrated with the Vienna Fortran Compilation System. Future work will investigate a tighter coupling of our analysis with target architectures without loosing too much portability, and possibilities to model cache interferences.

References

[1] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H.P. Zima. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[2] T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[3] Thomas Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47 – 56, November 1995.
Postscript file available via http://www.par.univie.ac.at/~tf/papers/p3t/ieee-mag.ps.

[4] J. Ferrante, V. Sarkar, and W. Trash. On estimating and enhancing cache effectiveness. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug 1991.

[5] K. Kennedy and K.S. McKinley. Optimizing for Parallelism and Data Locality. In *International Conference on Supercomputing 1992*, pages 323–334, Washington D.C., July 1992.

[6] M.E. Wolf and M. Lam. A data locality optimizing algorithm. In *In Proceedings of the SIGPLAN 91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.