Optimizing Inter-Nest Data Locality Using Loop Splitting and Reordering

Sofiane Naci

The Computer Laboratory, University of Cambridge
JJ Thompson Avenue
Cambridge CB3 0FD
United Kingdom
Sofiane.Naci@cl.cam.ac.uk

Abstract

With the increasing gap between processor speed and memory latency, the performance of data-dominated programs are becoming more reliant on fast data access, which can be improved using data locality optimization. Most studies in this area focus on optimizing data locality in individual loop nests. However, in many embedded applications, data access patterns exhibit a significant amount of inter-nest reuse. In this paper, we present a compiler strategy that optimizes inter-nest data locality using code restructuring and loop transformations. Our approach captures data reuse between all loop nests in the program and then splits and reorders the nests so that those sharing arrays are closer together. The transformed program is then further optimized using loop transformations. We improve on previous studies by using global program analysis and Integer Linear Programming to find the best nest ordering. The approach has been tested on many data-intensive embedded kernels and our simulation results indicate promising performance improvements.

1 Introduction

Array-intensive programs have been the target for optimization for many years. However, the majority of locality optimization techniques found in literature focus on improving locality in single loop nests which are optimized in isolation [14, 20]. This kind of optimization improves data locality within the loop nest while in most array-intensive applications, arrays are typically accessed in more than one nest and there exists major data reuse between loops accessing the same data. In their detailed study on the quantification of data locality in loop nests, McKinley and Temam [15] have shown that 80% of cache misses are inter-nest, i.e. occuring on data that was previously fetched to the cache in another loop nest. This clearly indicates that new techniques targeting locality optimization in multiple loop nests are needed.

While intra-nest optimization can improve data locality in each nest at a time, they fail to capture inter-nest data reuse. Optimizing multiple loop nests at a time requires more advanced analysis and informative representations of the program. Traditional techniques such as loop fusion [9, 13] and reversal [10] have been used in this area but their benefits are diminishing because of increasing program complexity. Global fusion algorithms also have some limitations. Kennedy and McKinley [11] use fusion and distribution to improve parallelism. However they do not handle fusion preventing dependencies and loops with non-identical iteration spaces. Ding [7] illustrates the use of loop fusion in reducing storage requirements through an example, but does not provide a general solution. More recently, Verdoolaege *et al.* [19] proposed a greedy algorithm for incremental loop fusion, using loop shifting to adjust the loop indexes. However, the dependence analysis performed using a graph model on a low level of abstraction may lead to unnecessary shifting and make the algorithm inefficient.

Inter-nest locality has also been directly targeted by [1] and [10]. In [1], Ahmed *et al.* propose a technique based on the generalization of multiple loop iteration spaces into a unified space, which is then optimized for better data locality. The main limitation to this approach is the fact that not all loop nests necessarily share arrays, which reduces the

1-4244-0910-1/07/$20.00 ©2007 IEEE.

scope of the approach and incurs major overheads. More recently Kandemir *et al.* [10] targeted the problem using loop reversal. Given two consecutive loop nests which access the same array, their approach applies reversal to a the second loop nest in order to take advantage of the elements left in the cache at the end of the execution of the previous loop. The approach is more suitable for programs composed of successive loops sharing the same data and it may not succeed in optimizing complex programs where many (adjacent) loop nests share more than one array, due to possible conflicts in array access patterns. These issues have been addressed in this paper.

To demonstrate the idea, consider the example shown in Figure 1.

```
for i = 1 to N do
  A[i] = i + 2
  D[i] = i

for j = 1 to N do
  C[j - 1] = j

for k = 1 to N do
  s += A[k]
  B[k] = k + 2
  D[k] = D[k - 1] + 1
```

(a)

```
for i = 1 to N do
  A[i] = i + 2
  D[i] = i

for j = 1 to N do
  C[j - 1] = j

for k = N to 1 do
  s += A[k]
  B[k] = k + 2
  D[k] = D[k - 1] + 1
```

(b)

```
for i = 1 to N do
  A[i] = i + 2

for k = 1 to N do
  s += A[k]

for i = 1 to N do
  D[i] = i

for k = 1 to N do
  D[k] = D[k - 1] + 1

for j = 1 to N do
  C[j - 1] = j

for k = 1 to N do
  B[k] = k + 2
```

(c)

Figure 1. Running example program. (a) Original. (b) After loop reversal [10]. (c) After loop splitting and reordering

Program (a) accesses four arrays in three loops. Arrays A and D are shared between loops 1 and 3 and are accessed using the same pattern. The program can be optimized using Kandemir *et al.*'s approach [10] as shown in Figure 1(b). The third loop is reversed such that elements of arrays A and D are accessed before being replaced in the cache. This would work well on simpler programs where data is shared by adjacent loop nests. However, in this example, loops sharing the same arrays are further apart in the program and Kandemir's approach returns diminishing results as will be empirically shown in Section 4. Assuming the cache is smaller than the total size of the arrays in the program, this is mainly due to cache capacity misses caused by accesses to array C in the middle loop nest, which fills the cache causing elements of the arrays A and D to be replaced before being reused in the third loop nest.

To overcome this problem and take better advantage of inter-nest reuse, it is important to analyze how different arrays are shared among different loop nests in the whole program. In order to improve locality between two loop nests sharing the same data, we need to (1) allow a maximum amount of the data to be in the cache between the execution of the two nests; and (2) allow the second loop nest to use this data while it is still in the cache.

In this paper, we present an inter-nest data locality optimization technique based on loop splitting and reordering to place loops referencing the same arrays closer together. An example is shown in Figure 1(c) where the nests in Figure 1(a) have been split and the resulting nests reordered. The transformed program in then further enhanced by loop transformations. Previous studies such as [6] and [11] also adopted loop scheduling for fusion, but using graph formulations. The problem of loop scheduling is a very special case of scheduling Directed Acyclic Graphs (DAGs), which is known to be NP complete [18]. As a result, solutions based on Integer Linear Programming (ILP) can be practical, especially if the number of variables and constraints remain small.

The main contributions and limitations of this work are:

- We introduce loop splitting and reordering as a program transformation to improve inter-nest locality.
- We measure inter-nest locality by analyzing array accesses in all the loop nests of the program at procedure level. We generalize the *nest distance*, previously introduced in [15], and use it as our cost function.
- We formulate the loop nest reordering problem in Integer Linear Programming (ILP) to find the best loop ordering, taking into account data dependencies.
- We evaluate the effectiveness of our approach using a set of array-intensive applications from the embedded domain and show how the approach can be integrated with loop reversal and fusion for better results.
- In this study, we assume loop nests are rectangular in shape in order to allow loop reversal. Furthermore, we assume that there are no array accesses contained within conditional statements. Extensions to the work are discussed in Section 5.

The remaining of this paper is organized as follows. In Section 2 we present our optimization technique and explain how to formulate the loop reordering problem in ILP. In Section 3 we show how the approach is used to optimize inter-nest locality. In Section 4 we outline our experimental results. Finally, Section 5 concludes the paper and summarizes the ongoing and future work.

2 Loop Splitting and Reordering

2.1 The Nest Distance

An inter-nest cache miss is a miss on data that was previously fetched in another loop nest. When optimizing for inter-nest locality, the natural question that arises is how far apart the nests are in the program. We quantify this information by calculating the global *nest distance*.

Definition 1. *The global nest distance in a program is the total number of loop nests separating each two loops referencing the same array.*

Two references occurring in the same loop nest have nest distance 0 while two references occurring in adjacent nests have a nest distance 1. The nest distance is calculated for each array in the program individually. Assume a program containing n loop nests $L_1, \ldots, L_n$ and accessing m arrays $A_1, \ldots, A_m$. The nests are numbered r_1 to r_n, where r_i is the order of L_i in the program for $1 \leq i \leq n$. Initially, $r_i = i$.

Definition 2. *Two loop nests L_i and L_j are said to share an array A, denoted $L_i \overset{A}{\Longleftrightarrow} L_j$, if L_i and L_j access elements of A.*

Definition 3. *The nest distance of an array A in a program is the total number of loop nests separating each two loops sharing A.*

The nest distance d_x of array A_x is calculated as:

$$d_x = \sum_{i,j=0}^{n} r_{i,j} \text{ where}$$

$$r_{i,j} = \left\{ \begin{array}{ll} r_i - r_j & \text{if } L_i \overset{A_x}{\Longleftrightarrow} L_j \text{ and } r_i > r_j \\ 0 & \text{otherwise} \end{array} \right\}$$

The total nest distance in the program, $\mathcal{D}$, is the sum of the nest distances of all arrays:

$$\mathcal{D}(L_1, \ldots, L_n) = \sum_{x=1}^{m} d_x$$

2.2 Motivation

When two loop nests that share a certain array are far apart in the program, i.e. the shared array has a large nest distance, elements of the array are more likely to be evicted from the cache by the time control reaches the second loop nest, which results in poor inter-nest locality. This issue is overcome by minimizing the nest distances of shared arrays. Studies like [15] have shown that because of cache organization, more than 60% of inter-nest reuse occur within short nest distances of at most 4, and more than 25% of inter-nest misses are within nest distance 1, i.e. between adjacent nests, assuming the nests share at least one array. This clearly indicates the importance of reducing the nest distance in order to take full advantage of inter-nest data reuse. The nest distance is reduced by loop splitting and reordering. This transformation minimizes cache capacity misses and increases the potential of inter-nest data locality optimization.

2.3 Loop Splitting

Loop splitting is not always legal and may be bound by data dependencies between statements in the loop body. In order not to harm temporal and spatial locality within the loop nest, loop splitting aims to separate disjoint statement only, i.e. statements within the loop nest that use disjoint sets of data. These are placed in separate loop nests.

To illustrate the idea, consider again the original example shown in Figure 1(a). After applying loop splitting, the transformed program is shown in Figure 2(a). The program now contains six loops accessing a single array each.

Loop splitting used in this context is an important step of the transformation. It prepares the program for loop reordering and further loop optimizations. In addition to this, where the original program contains imperfectly nested loops, the number of these is reduced by loop splitting, which results in better gains by loop transformations such as fusion.

2.4 Loop Reordering Using ILP

The aim of loop reordering is to localize data in the program by bringing loops that share the same arrays closer together. This is modeled and solved using Integer Linear Programming (ILP). The task is to assign new order values to loop nests such that the objective function is minimized, while maintaining a set of constraints. We use the nest distance as the objective function, and use constraints to (1) express data dependencies, for example indicate that a given loop nest must execute before another one; (2) express the uniqueness of each order value assigned to a loop nest; and (3) specify the range of possible order values.

Consider a program with k loop nests $L_1, \ldots, L_k$. Let r_i be the order of L_i in the program for $1 \leq i \leq k$. Initially, $r_i = i$. The aim of nest reordering is assigning a new order value from the range $(1, k)$ for each r_i with $1 \leq i \leq k$,

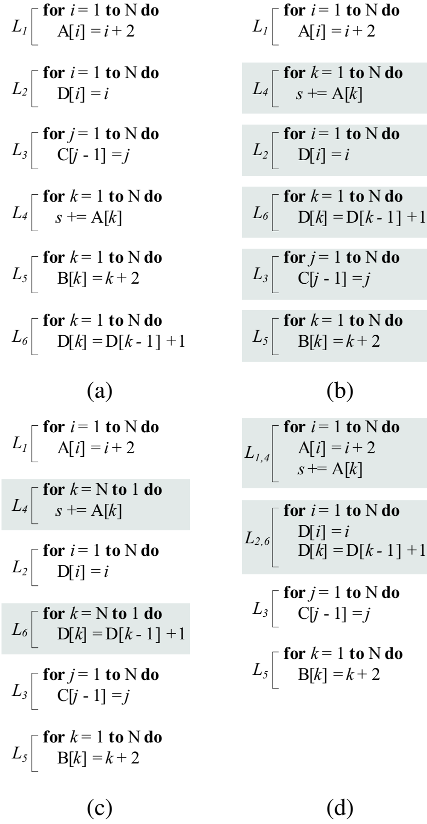

Figure 2. Running example program. (a) After loop splitting. (b) After loop splitting and reordering. (c) Combining with loop reversal. (d) Combining with loop fusion

such that the nests accessing shared arrays are brought closer together.

Solving the problem in ILP is illustrated through the example shown in Figure 2(a). The program contains six nests L_1, L_2, L_3, L_4, L_5 and L_6 with order values 1, 2, 3, 4, 5 and 6 respectively. We have $L_1 \overset{A}{\Longleftrightarrow} L_4$ and $L_2 \overset{D}{\Longleftrightarrow} L_6$, so

$$\mathcal{D}(L_1, L_2, L_3, L_4, L_5, L_6) = (r_4 - r_1) + (r_6 - r_2)$$

Due to data dependencies, L_1 and L_2 must execute before L_4 and L_6 respectively. So we formulate the reordering problem as:

$$\begin{array}{rl} \textit{minimize} & (r_4 - r_1) + (r_6 - r_2) \\ \textit{where} & r_1 < r_4 \text{ and} \\ & r_2 < r_6 \text{ and} \\ & (r_1, r_2, r_3, r_4, r_5, r_6) \text{ is a permutation} \\ & \text{of } (1, 2, 3, 4, 5, 6) \end{array}$$

which has an obvious solution $(r_1, r_2, r_3, r_4, r_5, r_6) = (1, 3, 5, 2, 6, 4)$. The corresponding program for this solution is shown in Figure 2(b). In general, the formulated problems are not large and can be computed efficiently. In our framework, we used the freely available ILP solver `lp_solve` [3].

Note that this step is concerned with array accesses and is independent on the structure or depth of the loops. Although imperfectly nested loop may still be present after loop splitting, they will not affect the reordering. In addition to this, since a given array is typically accessed in loops having the same depth, the reordering will also result in grouping loops having the same depth together.

3 Inter-nest Locality Optimization

Loop splitting and reordering presented in the previous section is a powerful enabling transformation for optimizing inter-nest data locality using loop transformations. In this paper, we consider the combination of loop fusion and loop reversal, which have been traditionally used for this matter. Other loop transformations, such as skewing and strip mining, are also known to result in performance gains and will be considered in future extensions to this work.

3.1 Combination with Fusion and Reversal

Inter-nest locality can be improved by cross loop transformations. The main works in this area include loop fusion [9, 13] and reversal [10]. Loop fusion enhances both temporal and spatial locality and is best used when the two loop bodies to be fused share the same array(s). Loop reversal on the other hand allows a loop nest to take advantage of data elements left in the cache after the execution of previous nests.

In this study, we investigate the benefits that loop fusion and reversal can add to the presented approach. Loop splitting and reordering increase the number of candidate loops for fusion and make it more beneficial, as the candidate loops are more likely to be accessing similar arrays. As for loop reversal, loop splitting and reordering reduce the number of capacity misses and therefore increase the benefits of loop reversal for inter-nest data locality optimization.

3.2 Optimization Algorithm

The optimization technique presented in this paper is summarized below:

1. Apply loop splitting on each loop nest to place disjoint statements in separate nests.

2. Apply loop reordering to minimize the nest distance of the program.
   (a) Formulate the ILP problem.
   (b) Solve the problem to find the new reordering.
   (c) Transform the program.
3. Apply loop fusion recursively.
4. Apply loop reversal using Kandemir's method [10].

Note that the third step may not be applied all the time due to data dependencies issues. The loop fusion algorithm used in this study is a simple one which fuses two candidate loop nests if there are no data dependencies between them and their loop headers are identical. The candidate nests must also be adjacent and access the same array(s). Loop fusion is applied recursively in order to allow more than two loops to be combined. Other details regarding the application of loop fusion and loop reversal are omitted due to the lack of space and can be found in [9, 13, 12].

4 Experiments

4.1 Setup

Version	Loop transformations applied
R	Reversal only, according to [10].
SR	Splitting + reordering, then Reversal.
SF	Splitting + reordering, then fusion.
SFR	SF version + reversal.

Table 1. Versions of the programs used in the experiments

Our inter-nest data locality optimization algorithm has been implemented using the SUIF 2 experimental compiler infrastructure [2]. We used eight embedded kernels to test the effectiveness of the algorithm; `fir`, an array version of the DSPStone's 2-D FIR filter [21]; `k14`, a C version of kernel 14 of the Livermore benchmark which implements a 2-D particle in cell algorithm [16]; `k18`, a C version of kernel 18 of the Livermore benchmark which calculates an explicit hydrodynamics fragment [16]; `lms`, an array version of the DSPStone's true LMS algorithm [21]; `iir`, a 1-D LMS adaptive second-order IIR filter [8]; `lnb`, a Linear Boundary Value problem solver [17]; `nlb`, a Non-Linear Boundary Value problem solver [17]; and `inv`, an eigenvalue problem solver using the inverse power method [17]. Using SUIF 2, we applied different combinations of the transformations described in Sections 2 and 3 and we generated four transformed versions of each original program as described in Table 1. The characteristics of each benchmark and details of its transformation are summarized in Table 2.

Benchmark	A	ADS	L	R	S	RS	SF
`fir`	6	200	5	2	8	3	4
`k14`	10	1000	4	1	8	4	6
`k18`	9	175	4	2	9	5	7
`lms`	2	128	5	1	8	3	5
`iir`	13	300	6	2	9	4	6
`lnb`	4	780	3	1	8	4	3
`nlb`	5	2000	5	1	10	4	5
`inv`	6	235	8	3	14	7	5

Key
A: The number of arrays
ADS: The approximate data size (KB)
L: The number of loops, originally
R: The number of reversed loops
S: The number of loops after splitting
RS: The number of reversed loops after splitting
SF: The number of loops after splitting + fusion

Table 2. The transformations performed on the benchmarks

The original benchmark programs and the transformed versions were compiled using a cross `gcc` compiler with the highest optimization level (`-O3`) to stress the analysis. The programs were then simulated using SimpleScalar's `sim-cache` simulator [4]. The simulator's memory hierarchy was configured as shown in Table 3. We use typical figures from the embedded system domain for the level-1 data cache, the level-2 unified cache and the data TLB (Translation Lookaside Buffer), where **C** denotes the capacity or the total size, **B** the block/page size, **Assoc.** the associativity and **Lat.** the access latency in clock cycles.

Part	C	B	Assoc.	Lat.
L1 D-cache	16 KB	32 B	4	1
L2 U- cache	256 KB	32 B	4	6
Data TLB	16 entries	4 KB	4	75
Main memory	N/A	N/A	N/A	150

Table 3. SimpleScalar's memory configuration

4.2 Results and Discussion

Using the simulator, we collect information about the memory hierarchy performance, calculate L1, L2 and TLB miss rates and compute the effective data access time EAT $= (1 - m)a + mp$ where m is the miss rate recorded at the L1 cache, a is the access latency of the L1 cache and p is the miss penalty of the L1 cache. In this case where we have two levels of cache, the miss penalty of the L1 cache is essentially the effective access time of the L2 cache. In our calculation, we also take into account the penalty of TLB misses. All the obtained results are expressed as percentage improvements with respect to the original code. Note that the reported improvements are observed within the main loop kernel, which is the most costly operation of the program. Improvements obtained by running the whole program have reached 90 to 95% of the reported figures.

Figure 3(a) shows the percentage improvements in L1 data cache miss rate. On average, we record miss rate improvements of 1.50, 1.84, 8.63 and 11.31% for the four transformed versions R, SR, SF and SFR respectively. This shows the benefits of loop splitting and reordering for inter-nest locality. It can be seen that in six out of the eight benchmarks used, the SR transformation of the program resulted in more improvements compared to the R transformation. While the difference is rather small, the results show that coupling loop splitting and reordering with loop fusion brings significant performance benefits and can reduce the L1 cache miss rate by up to 3 times.

Note that in two of the programs, namely `f14` and `iir` the SR version did not result in any improvements in terms of L1 data cache miss rate. This is mainly due to overheads incurred from having many more loop nests in the program compared to the original and R versions. To investigate this case further, Figure 3(b) shows the percentage improvements in miss rates averaged over all components of the memory hierarchy. This includes the L2 cache misses and the data TLB misses. The figure shows that even when there are no significant improvements recorded in the L1 data cache, other components of the memory hierarchy still benefit from the transformations.

To see the benefits of reduced miss rates in the memory hierarchy, we show in Figure 3(c) the percentage improvements in the effective data access time. The figure depicts significant reductions in data access time and shows that the R transformation is outperformed by the other transformations in all the benchmarks used. On average, we record improvements of 12.60, 16.90 and 22.35% for the SR, SF and SFR versions respectively while the average improvement gained by the R version is only 7.15%. All these results indicate that the presented approach outperforms the one presented in [10] when dealing with more complex programs.

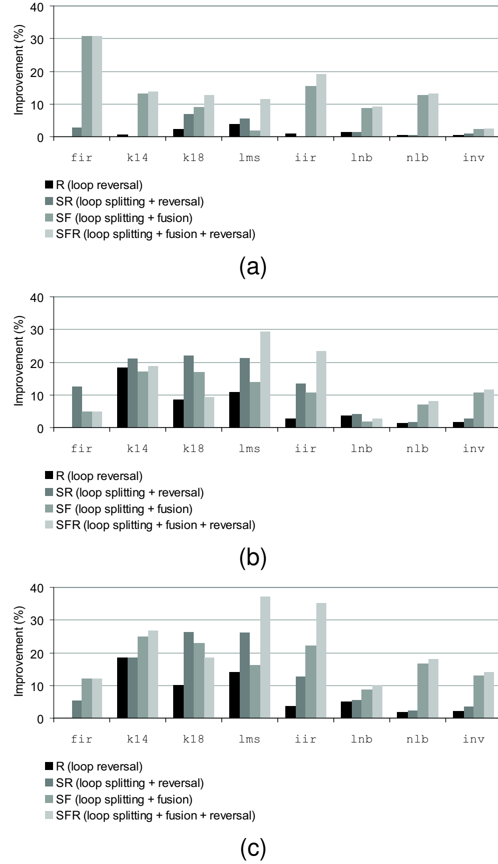

Figure 3. Percentage improvements in (a) L1 data cache miss rate (b) average memory system miss rate (c) effective data access time

4.3 Effects on Code Size

To investigate the effects of our approach on code size, we measured the size of our benchmarks after loop splitting and after loop splitting and fusion. The percentage changes with respect to the original program are illustrated in Figure 4. The figure shows that splitting increases the code size for all benchmarks, by up to 27%. However, applying loop fusion reduces it back to its original value in most cases. For some becnhmarks, we still record a slight (less than 5%) increase compared to the original code size. For others, fusion actually makes the code even smaller than originally. These figures indicate that although loop splitting increases the size of the code, it enables better loop fusion, which could make the code more compact than before.

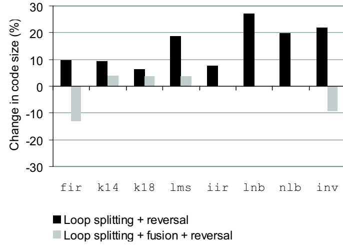

Figure 4. Effects of splitting and fusion on code size

4.4 Effects of Cache Size

In order to analyze the effects of cache size on the effectiveness of the approach, we run the same experiments described in Section 4.1 with different L1 cache sizes, while keeping the other memory hierarchy parameters unchanged. Figure 5 shows the average improvements recorded for all the benchmarks in the Effective Access Time and miss rates for different cache sizes. The first remark that can be drawn from the figures is that combining fusion and reversal always outperforms the other combinations. The figures also show that the simplest approach, consisting on loop reversal only, is the worst one in terms of performance benefits.

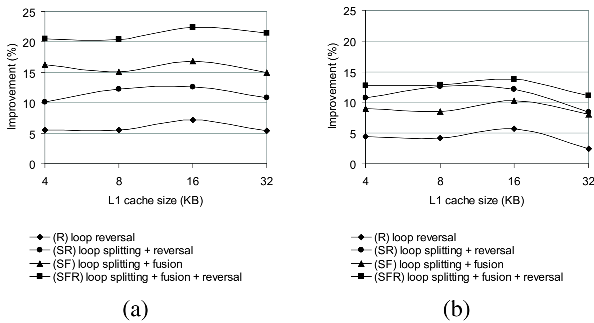

Figure 5. Effects of cache size (a) on effective data access time (b) on average miss rate

The level of improvements is highly dependent on the cache size. However, increasing the cache size does not necessarily result in more performance. To explain this, consider a simple example where a cache of capacity C is organized into blocks of size B. When an array of size N is accessed sequentially, the number of cache misses is N/B. If we access the same array in two different loop nests, we distinguish two cases. If $N < C$ the total number of misses is N/B since all accesses in the second nest hit in the cache. Note that in this case, inter-nest optimization does not bring any extra benefits. As a result, we focus on the case where $N > C$. In this case, we record $2N/B$ misses, where N/B of them are compulsory misses, due to loading data for the first time, and the remaining N/B are capacity misses due to data being evicted from the cache. By applying loop reversal on the second nest, we save C/B of the capacity misses recorded in the second nest. This therefore results in a benefit ratio of $C/2N$, reaching a maximum 50% improvement when $C = N$. This shows that with a small cache, i.e. C is small, the improvements become negligible as shown in Figure 5 for the cache sizes of 4 and 8KB. When C is much larger than N, this improves the performance of the original program and renders the improvements brought by the transformation small too. This is shown in figures for cache size of 32KB.

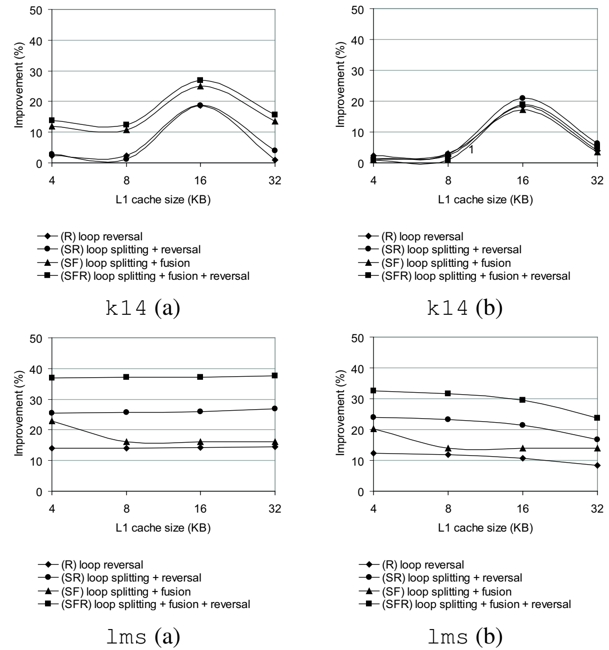

Figure 6. Effects of cache size on `k14` and `lms` (a) on effective data access time (b) on average miss rate

These changes in performance vary for each benchmark according to the size of the data and the length of the arrays handled by the program. Figure 6 shows, for example, the effects of changing the cache size on the performance of the `k14` and `lms` benchmarks. For `k14`, we observe dramatic changes in performance for different cache sizes. The improvements are small for small and large sizes and significantly better for a 16KB cache. For `lms` however, we observe a more constant improvement ratio across dif-

ferent sizes with slight degradation for large caches. The main reason for these differences is the amount of data handled by the program. As shown in Table 2, `k14` uses 10 arrays accessed in several loop nests which causes more cache misses and makes the effects of the transformation with respect to the cache size more visible. On the other hand `lms` uses only 2 arrays which reduces the number of cache misses and leaves a smaller room for optimization compared to `k14`.

5 Conclusions and Future Work

In this paper we have demonstrated that there exists a significant amount of performance benefits that can be obtained by inter-nest data locality optimization. The algorithm proposed in this paper targets this point and achieves significant improvements when combined with existing loop transformations. Our experiments show that our framework improves cache miss rate by up to 31%, leading to a 38% faster data access time. However, the approach can be improved in many ways. The cost function introduced in this paper is simple and may lead to suboptimal results in special cases when, for example, two loop nests share an array but access non-overlapping sections of it. This problem can be overcome by introducing weights when calculating the nest distance, related to the number of array elements that are effectively shared between the nests. This is the subject of current ongoing extensions. We are also aware that the effectivness of the approach highly depends on the size of the cache and the arrays in the program. This limitation will be overcome by combining the approach with simple data transformations that change the size of arrays such as array composition/decomposition [5] and other loop transformations like strip-mining. In addition to the above, future studies will investigate how to generalize the approach to optimize inter-procedural locality.

References

[1] M. Ahmed, N. Mateev, and K. Pigalli. Synthesizing Transformations for Locality Enhancement of Imperfectly-Nested Loop Nests. In *Proceedings of the 14th International Conference on Supercomputing*, pages 141–152, 2000.

[2] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An Overview of the SUIF2 Compiler Infrastructure. Technical report, Stanford University, 2000.

[3] N. Berkelaar. lp_solve Mixed Integer Linear Programming solver, version 5.5.0.8. ftp://ftp.es.ele.tue.nl/pub/lp_solve.

[4] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, 1997.

[5] G. Chen, M.Kandemir, U. Sezer, and A. Nadgir. Array Composition and Decomposition for Optimizing Embedded Applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2003.

[6] A. Darte. On the Complexity of Loop Fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[7] C. Ding. *Improving Effective Bandwidth Through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, 2000.

[8] P. Embree. *C Algorithms for Real-Time DSP*. Prentice Hall, 1995.

[9] A. Fraboulet, K. Godary, and A. Mignotte. Loop Fusion for Memory Space Optimization. In *Proceedings of the International Symposium on System Synthesis*, pages 95–100, 2001.

[10] M. Kandemir, I. Kadayif, A. Choudhary, and J. Zambreno. Optimizing Inter-Nest Data Locality. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 127–135, 2002.

[11] K. Kennedy and K. S. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, 1994.

[12] R. Leupers. Code Generation for Embedded Processors. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 173–178, 2000.

[13] P. Marchal, J. I. G'omez, and F. Catthoor. Optimizing the Memory Bandwidth with Loop Fusion. In *Proceeding of the International Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[14] K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming and Systems*, 18(4):424–453, 1996.

[15] K. McKinley and O. Temam. Quantifying Loop Nest Locality Using SPEC'95 and the Perfect Benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, 1999.

[16] F. McMahon. The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[17] S. Nakamura. *Applied Numerical Methods in C*. Prentice Hall International Editions, 1993.

[18] J. Ullman. NP-complete Scheduling Problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[19] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-Dimensional Incremental Loop Fusion for Data Locality. In *Proceedings of the IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, 2003.

[20] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[21] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the Conference on Signal Processing Applications and Technology*, 1994.