Compiler-Assisted Cache Replacement: Problem Formulation and Performance Evaluation*

Hongbo Yang[1], R. Govindarajan[2], Guang R. Gao[1], and Ziang Hu[1]

[1] Department of Electrical and Computer Engineering, University of Delaware
Newark, DE 19716, USA {hyang,ggao,hu}@capsl.udel.edu
[2] Department of Computer Science and Automation, Indian Institute of Science
Bangalore, 560012, INDIA govind@csa.iisc.ernet.in

Abstract

Recent research results show that conventional hardware-only cache solutions result in unsatisfactory cache utilization for both regular and irregular applications. To overcome this problem, a number of architectures introduce instruction hints to assist cache replacement. For example, Intel Itanium architecture augments memory accessing instructions with cache hints to distinguish data that will be referenced in the near future from the rest. With the availability of such methods, the performance of the underlying cache architecture critically depends on the ability of the compiler to generate code with appropriate cache hints. In this paper we formulate this problem – giving cache hints to memory instructions such that cache miss rate is minimized – as a 0/1 knapsack problem, which can be efficiently solved using a dynamic programming algorithm. The proposed approach has been implemented in our compiler testbed and evaluated on a set of scientific computing benchmarks. Initial results show that our approach is effective on reducing the cache miss rate and improving program performance.

1 Introduction

Over the last few decades, as the processor performance kept undergoing substantial progress, the gap between processor and memory speeds has been widening steadily. This problem, known as the "memory wall" problem, exists in both general-purpose high-performance computers [13] and embedded systems [17]. To bridge this performance gap, cache is introduced which has ameliorated the "memory wall" problem to some extent. However, a conventional cache is typically designed in a hardware-only fashion, where data management including cache line replacement is decided purely by hardware. A consequence of this design approach is that cache can make poor decisions in choosing data to be replaced, which may lead to poor cache performance. The widely used LRU (least recently used) cache replacement algorithm makes replacement decisions based on past reference behavior. This can cause data with good reuse yield cache space to data that comes in later but has poor reuse. Research results reveal that considerable fraction of cache lines are held by data that will not be reused again before it is

* This research is funded by NSF, under the NGS grant 0103723, DOE, grant number DE-FC02-01ER25503 and Intel Corp.

displaced from the cache. This is true for both irregular [4] and regular applications [15]. This phenomenon, called *cache pollution*, severely degrades cache performance.

There are a number of efforts in architecture design to address this problem and the *cache hint* mechanism implemented in the Intel Itanium processor [9] is one of them. The memory accessing instructions of Itanium can be accompanied by a `nt` (stands for *non-temporal*) cache hint. In response, Itanium-2 implemented a modified LRU replacement algorithm honoring the `nt` cache hint [9]. In the Itanium-2 processor, the execution of memory accessing instructions with `nt` cache hint differs from that of a normal memory instruction in the following way. For a set-associative cache, when a normal memory instruction is executed, a cache line is allocated for the accessed data, and the just allocated cache line is given the highest rank in the set (to indicate that it is the most recently used). Thus it becomes the last to be replaced among all cache lines in the particular set. In contrast, the execution of a memory instruction with `nt` cache hint does not change the rank of the touched cache line. In this modified LRU replacement mechanism, data accessed by instructions with `nt` hint is more likely to be evicted on a subsequent cache miss. By relying on the compiler to give `nt` hint to the instructions accessing data without temporal reuse, this architecture effectively prevents cache pollution thus has the potential to achieve better cache locality. On this architecture, a good compiler algorithm to generate cache hint is essential, which is the focus of this paper.

Intuitively, two kinds of memory instructions should be given `nt` hint: (i) whose referenced data doesn't exhibit temporal-reuse. (ii) whose referenced data does exhibit temporal-reuse, but it cannot be *realized* under the particular cache configuration. It sounds as though the problem is pretty simple for regular applications, and existing techniques for analyzing data reuse [20] and estimating cache misses [11, 21, 12] suffice to solve this problem. This plausible statement, however, is not true because a fundamental technique used in cache miss estimation — footprint analysis — is based on the assumption that all accessed data compete for cache space equally. However, in our target architecture, memory instructions are not homogeneous — those with cache hints have much less demand for cache space. This makes the approach derived from traditional footprint analysis very conservative. In summary, the following *cyclic dependence* exists: Cache hint assignment must be known to achieve accurate cache miss estimation, while accurate cache miss estimation is only possible when cache hint assignment is finalized.

In this paper, we develop a simple yet effective formulation to address the above problem. Our formulation is based on the observed relationship between cache miss rate and cache-residency of *reference window* [10]. This is used to formulate the problem as a 0/1 knapsack problem [8]. For the case that all considered memory referencing instructions are enclosed by a *perfect loop nest*, the formulated problem falls in a special category of knapsack problem that can be solved in polynomial time. For case that loops are imperfectly nested, this is a general 0/1 knapsack problem, which is known to be NP-complete [8]. In this case, good heuristic algorithms exist to achieve near-optimal result [5]. However, since the number of references in a loop nest is typically small, even obtaining optimal result using a dynamic programming algorithm [8, 14] is quite inexpensive.

We have evaluated the benefit of our approach on reducing cache misses on a set of loop kernels and a full SPEC benchmark program by simulating their execution using the SimpleScalar simulator [3]. Initial experimental results show that our approach reduces the number of data cache misses by up to 57.1%, and reduces execution time by up to 27%.

The rest of the paper is organized as follows. Section 2 briefly reviews the basic concepts of data reuse and reference window. Section 3 illustrates, through an example, the relationship between reference window and cache miss rate which sets up the rationale for our problem formulation. The heart of this paper — an elegant knapsack problem formulation — is derived in Section 4. Our implementation and experimental results are then presented in Section 5. Section 6 discusses related work. Section 7 concludes the paper and envisions possible future research directions.

2 Preliminaries

We review some basic concepts on data reuse and reference window that will be used in the rest of this paper.

For an affine array reference in a loop nest of depth n, the subscripts can be represented as $H \cdot \overline{i} + \overline{c}$ (where H is the *access matrix*, $\overline{i}$ is the *iteration vector* and $\overline{c}$ is the *offset vector*). If two different executions of an array reference at iteration points $\overline{i_1}$ and $\overline{i_2}$ access the same array element, it must be true that $H \cdot (\overline{i_2} - \overline{i_1}) = 0$. Therefore, if the equation $H \cdot \overline{i} = 0$ has a solution, the array reference with subscripts $H \cdot \overline{i} + \overline{c}$ exhibits *self-temporal reuse* and the solution to $H \cdot \overline{i} = 0$ constitutes the *self-temporal reuse vector*. Two references to the same array, with the same access matrix but different offset vectors, say reference $H \cdot \overline{i} + \overline{c_1}$ and reference $H \cdot \overline{i} + \overline{c_2}$, may access the same data only if equation $H \cdot (\overline{i_1} - \overline{i_2}) = \overline{c_2} - \overline{c_1}$ can be satisfied. Thus *group-temporal reuse* exists when $H \cdot \overline{i} = \overline{c_2} - \overline{c_1}$ has a solution, and the solution constitutes the *group-temporal reuse vector*.

A *uniformly generated reference set* (UGS) is a set of references of the same array, with the same access matrix and has group data reuse within the set [10]. By defining uniformly generated reference set and partitioning all array references into UGSs, we can study data reuse on a per-UGS basis.

Gannon et al's work introduced the term *reference window*, which is defined as the set of array elements that are accessed by the source reference of a reuse-pair in the past and will be accessed in the future by the sink reference [10]. Consider the Fortran program shown in Figure 1 as an example. This is a small kernel from the `SPEC92` benchmark *093.nasa7*. Reference windows associated with all loop-carried reuse-pairs are listed in Figure 2.

Let us explain why the reference windows are as given in Figure 2. For reference **C(I,K)** at iteration (j, k, i), where $j > 1$ and $j < M$, the entire array has been traversed by previous iterations, and all the array elements will be accessed again, before the loop execution advances to $(j + 1, k, i)$. Therefore the reference window is the entire array. For the self-reuse of array reference **A(I,J)** at iteration (j, k, i) where $k > 1$ and $k < N$, all elements in the first dimension will be referenced in the future. Other reference windows given above can be derived similarly.

```
      DO 110 J = 1, M, 4
         DO 110 K = 1, N
            DO 110 I = 1, L
               C(I,K) = C(I,K) + A(I,J) * B(J,K)
            $  + A(I,J+1) * B(J+1,K) + A(I,J+2) * B(J+2,K)
            $  + A(I,J+3) * B(J+3,K)
110   CONTINUE
```

Fig. 1. The MXM loop kernel from SPEC92. Values of M, N and L are 128, 64 and 256 respectively, A, B and C are two dimensional arrays with 8-byte double precision floating-point array elements

$$
\begin{array}{ll}
\text{Ref\_Win}(\text{C(I,K)} \rightarrow \text{C(I,K)}) & = \{\text{C(1,1), C(1,2)} \cdots \text{C(1,N)}, \\
& \quad \cdots \\
& \quad \text{C(L,1),C(L,2)} \cdots \text{C(L,N)} \} \\
\text{Ref\_Win}(\text{A(I,J)} \rightarrow \text{A(I,J)}) & = \{\text{A(1,J), A(2,J),} \cdots \text{A(L,J)}\} \\
\text{Ref\_Win}(\text{A(I,J+1)} \rightarrow \text{A(I,J+1)}) & = \{\text{A(1,J+1), A(2,J+1),} \cdots \text{A(L,J+1)}\} \\
\text{Ref\_Win}(\text{A(I,J+2)} \rightarrow \text{A(I,J+2)}) & = \{\text{A(1,J+2), A(2,J+2),} \cdots \text{A(L,J+2)}\} \\
\text{Ref\_Win}(\text{A(I,J+3)} \rightarrow \text{A(I,J+3)}) & = \{\text{A(1,J+3), A(2,J+3),} \cdots \text{A(L,J+3)}\} \\
\text{Ref\_Win}(\text{B(J,K)} \rightarrow \text{B(J,K)}) & = \{\text{B(J,K)}\} \\
\text{Ref\_Win}(\text{B(J+1,K)} \rightarrow \text{B(J+1,K)}) & = \{\text{B(J+1,K)}\} \\
\text{Ref\_Win}(\text{B(J+2,K)} \rightarrow \text{B(J+2,K)}) & = \{\text{B(J+2,K)}\} \\
\text{Ref\_Win}(\text{B(J+3,K)} \rightarrow \text{B(J+3,K)}) & = \{\text{B(J+3,K)}\}
\end{array}
$$

Fig. 2. Reference windows for reuse pairs

A careful study reveals that the size of a reference window is determined by its reuse vector. By solving reuse equations $H \cdot \bar{i} = 0$ for each reference, we get the reuse vectors for references **C(I,K)**, **A(I,J)**, **B(J,K)** as $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ respectively. By the definition of reuse vector, we know that **C(I,K)** accesses the same element at iterations (j, k, i) and $(j + 1, k, i)$; however, these two iterations are far apart, thus the number of different array elements accessed in between (i.e., the reference window) is large. While the reuse of **B(J,K)** happens at the innermost loop, its reference window is much smaller. Gannon et al., gave a formula to compute the size of reference window based on reuse vector; we refer interested readers to [10] for more details.

3 A Case Study

In this section, using the matrix multiply program shown in Figure 1, we illustrate the relationship between reference window and cache miss rate.

First let us analyze the data reuse[3] for this program. We start with the array reference **A(I,J)** , data accessed by this reference at iteration (j, k, i) is $A(i, j)$ and it will be

[3] Data reuse is a term different from cache locality; data reuse leads to cache locality only when the reuse can be *realized* by the particular cache configuration.

accessed again by the same array reference at iteration $(j, k + 1, i)$. Intervening data accesses by all array references during this interval (from (j, k, i) to $(j, k+1, i)$) do not interfere with *A(i,j)*. This kind of reuse is named *self-reuse* [20]. Following the reuse analysis method given by Wolf and Lam [20] we can easily derive that types of data reuse of all other array references of **A** are self-reuse (there doesn't exist reuse between references **A(I,J+1)** and **A(I,J)** since the stride of loop J is 4).

Since there does not exist data reuse relation between any two references of **A**, we can study each reference of **A(I,J)**, **A(I,J+1)**, **A(I,J+2)** and **A(I,J+3)** in isolation. Without loss of generality, we choose the reference **A(I,J)** and profiled its cache behavior. Before giving the profiling result, we define the term *cache occupancy* to refer to the number of cache lines occupied by a particular array reference. We traced the cache occupancy and the cache miss rate for the reference **A(I,J)** on a 256-set, 4-way associative cache with a cache line size of 8 bytes. Both cache occupancy and cache miss rate are shown in Figure 3. In this figure, both cache occupancy and cache miss rate are obtained by averaging the respective values for the last 20 clock cycles. In the figure cache occupancy of the reference varies slightly from 255.5 to 256, while the cache miss rate varies widely, from 0% to 100%.

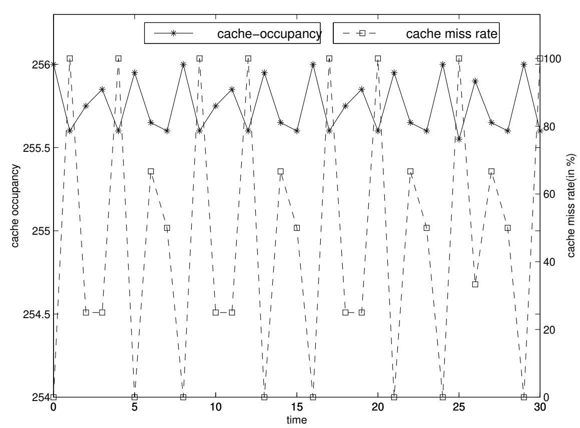


Fig. 3. Cache occupancy and miss rate of array reference **A(I,J)** in the program shown in Figure 1

We observe that the cache miss rate is tightly coupled with the cache occupancy, and is inversely proportional to cache occupancy. When the average cache occupancy of the reference is 256 for the last 20 cycles, the cache miss rate is zero during this period. While the cache occupancy reduces to 255.5 - 255.6 (due to competition with other array references), the cache miss rate rises to 100%. This is somewhat surprising, at least initially, as the decrease in the cache occupancy is only marginal (from 256 to 255.5).

Let us go back to the source program and analyze why this happens. As we have discussed before, the array element accessed by reference **A(I,J)** at iteration (j, k, i)

will be accessed again by the same array reference at iteration $(j, k+1, i)$. The number of distinct array elements accessed by **A(I,J)** in between (including the two bounding iterations) is 256. These 256 array elements are the reference window for the self-reuse vector of **A(I,J)** that we derived in Section 2. Hence we conclude that if the cache holds all elements of the reference window for a particular reuse pair, the data-reuse is translated into cache-locality at run-time; otherwise, that reuse cannot be exploited by the cache. Based on this observation, we formulated the problem of giving `nt` cache hint in Section 4.

4 Problem Formulation

In this section we give a problem formulation for generating `nt` hint for memory instructions. We start with the case that all memory references have self-reuse only and give the problem formulation in Section 4.1. The general case that includes group-reuse is discussed in Section 4.2.

4.1 Problem Formulation for Self-Reuse: Case I

The particular problem that we address in this section is as follows:
Problem 1. Given a cache size and a perfect loop nest whose loop body has m array references with no two references having data reuse between them, determine the subset of references that should be given `nt` hint such that cache miss rate of executing this loop nest on the given cache is minimized.

As demonstrated by the profiling result of matrix multiply program (shown in Figure 3), to realize a data reuse, the reference window of that data reuse must be accommodated by the cache. In reality, cache size is limited and reference windows that it can hold is subject to the cache capacity. We associate each array reference with a binary variable b_i to denote whether it is given `nt` hint($b_i = 0$) or not($b_i = 1$), the variables $b_1 \cdots b_m$ constitute all decision variables of the problem. The constraint imposed by cache capacity can be formulated as:

$$\sum_{i=1}^{m} |\text{Ref\_Win}(i)| * b_i < C \tag{1}$$

where $|\text{Ref\_Win}(i)|$ refers to the size of the reference window of array reference i, and C is the *effective cache size* [11, 18]. We use the *effective cache size* instead of *full cache size* in the capacity constraint since stride access with a stride larger than 1 cannot exploit the full cache capacity, as shown in Gao et al's work [11].

The capacity constraint ensures that for array reference i whose corresponding decision variable b_i has a value 1, its reference window will be fully accommodated by the cache. Hence its temporal reuse can be realized. Since our objective is to minimize the cache miss rate, it is desirable to have as many array references as possible achieve temporal locality. And since all array references are enclosed by a perfect loop nest, their execution frequencies are the same. Thus our objective function is formulated as:

$$\max \sum_{i=1}^{m} b_i \tag{2}$$

This problem composed of the constraint specified by Inequality 1 and the objective function (specified by Equation 2). This is, in essence, a 0/1 knapsack problem[8]. For the problem formulation that we have given, the knapsack problem falls into a special category where the candidate items have different *weights*(size of the reference window) but the same *value*(1). For this special case, the knapsack problem can be solved using a greedy algorithm in polynomial time. We give the details of such an algorithm in [22]. For more complicated cases where the loops are imperfectly-nested, the coefficients of b_i in the objective function will not be uniform, resulting in a more general 0/1 knapsack problem. For the general 0/1 knapsack problem, optimal result can be obtained by using a dynamic programming algorithm in $O(mC)$ time [8, 14], where m is the number of array references and C is the effective cache size. If the time-complexity of the dynamic programming approach is unaffordable, heuristic algorithm also exists to obtain near-optimal result [5].

4.2 Problem Formulation for Group Reuse: Case II

Now we extend our approach to the general case that group-reuse exists. The problem that we address in this section is:

Problem 2. Given a cache size and a perfect loop nest whose loop body has m array references that have group data reuse, determine the subset of references that should be given `nt` hint such that cache miss rate of executing the loop nest is minimized.

To address this problem, group reuse of these m array references should be figured out first. Then we can formulate this problem in a similar way as in Case I. Our approach to address this problem is therefore divided into the following three steps:

1. Partition the array references into UGSs.
2. Represent the reuse within each UGS using a *reuse graph* and prune the edges of the reuse graph to simplify the problem.
3. Form a 0/1 knapsack problem from the pruned reuse graph.

We illustrate these steps by using an example program:

```
    DO 10 T = 1, IT
       DO 10 I = 1,M
          DO 10 J = 1,N
             L(I,J) = (A(I,J-1) + A(I,J+1) + A(I-1,J) + A(I+1,J)) / 4
10  CONTINUE
```

Step 1. Partitioning: In the first step, we partition array references into UGSs such that group reuse exists only within each set. This step is the same as that documented in Wolf et al's paper [21] and Mowry's dissertation [16]. For the example program, the five array references are partitioned into two UGSs:

Set$_1$ = { **L(I,J)** }
Set$_2$ = { **A(I,J-1), A(I,J+1), A(I-1,J), A(I+1,J)** }

Step 2. Pruning: The nice feature of the target loops of Problem 1 that we dealt with in Section 4.1 is that data reuse is within each single reference, thus the cost and benefit of realizing the reuse is clearly defined. The presence of group-reuse makes this feature disappear and we have to deal with the case that data accessed by one array reference is reused by several other array references. We represent group data reuse using a *reuse graph* (as shown in Figure 4), where each edge (solid or dashed) represents a possible reuse. The reuse graph can be simplified such that each reference has only one successor and one predecessor. In the following paragraph we discuss how to prune the reuse graph. In Figure 4, edges remaining after pruning are shown as solid edges and edges that can be pruned are shown as dashed edges. For legibility reasons, we did not show all pruned edges. However all solid edges that remain after pruning are shown.

Consider the reuse between **A(I+1,J)** and **A(I,J-1)** as an example. Although reuse testing by solving the reuse equation renders us a reuse edge from **A(I+1,J)** to **A(I,J-1)**, a careful analysis reveals this reuse actually does not happen. This is because of the intervening access generated by **A(I,J+1)**. Consider the location *A(i+1,j)* accessed by references **A(I+1,J)** and **A(I,J-1)**. The above accesses happen respectively at iteration (i, j) and $(i + 1, j + 1)$. Before this reuse can be realized between these two references, a reuse by reference **A(I,J+1)** happens at iteration $(i+1, j-1)$. Hence the reuse edges (**A(I+1,J)**, **A(I,J+1)**) and (**A(I,J+1)**, **A(I,J-1)**) together, transitively, represent the reuse information between **A(I+1,J)** and **A(I,J-1)**. Therefore the edge (**A(I+1,J)**,**A(I+1,J-1)**) can be pruned. In a similar way, all transitive edges can be pruned from the reuse graph. By pruning the transitive edges, we get a reuse graph in which each node has at most one successor and one predecessor. This nice feature of the pruned reuse graph facilitates our knapsack problem formulation since the cost and benefit of realizing each reuse can be easily identified.

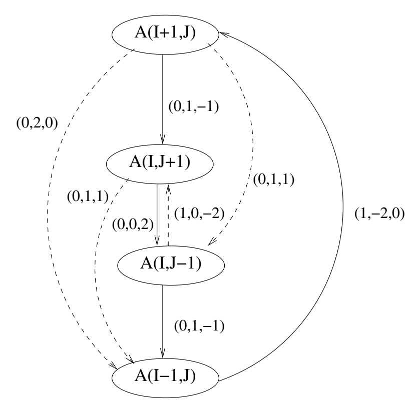


Fig. 4. Data reuse graph for Set_2 of the example program. The vector adjacent to each reuse edge is the reuse vector

As seen, for multiple array references that possibly reuse data of a common parent, the pruning step chooses the one that reuses the data at the earliest time. Thus the rule for pruning is: For an array reference which emanates multiple reuse edges, keep the edge that has minimum reuse vector and prune all other edges. Reuse vectors are ordered in *lexicographic order* [1].

Step 3. Formulation: After pruning we proceed to the last step, viz., formulating the problem. The cost of realizing a temporal reuse is size of the reference window associated with the reuse. By realizing the reuse, the reference reusing the data will get its data from cache instead of memory, thus saving a memory reference for an iteration. In the pruned reuse graph, the reference window size of the four reuse edges emanating from **A(I+1,J), A(I,J+1), A(I,J-1) and A(I-1,J)** are $N-1, 2, N-1$ and $(M-2)\cdot N$ respectively. **L(I,J)** has self-reuse with reference window size of $M \cdot N$. The problem for the example program can be formulated as:

Maximize:

$$b_{A(I+1,J)} + b_{A(I,J+1)} + b_{A(I,J-1)} + b_{A(I-1,J)} + b_{L(I,J)}$$

within the constraint:

$$(N-1)\cdot b_{A(I+1,J)} + 2\cdot b_{A(I,J+1)} + (N-1)\cdot b_{A(I,J-1)} + \\ (M-2)\cdot N\cdot b_{A(I-1,J)} + M\cdot N\cdot b_{L(I,J)} < C$$

5 Experimental Results

5.1 Experimental Platform

We have implemented our approach in the MIPSpro compiler and evaluated its performance by running SPEC benchmarks on SimpleScalar simulator [3]. The MIPSpro compiler is the production-quality compiler developed by SGI for MIPS processors. We have re-engineered the code generator of the MIPSpro compiler to generate code for SimpleScalar. The MIPSpro compiler has a rich set of optimizations to maximize program performance. We have enabled most of them in our experiment. As a first step of our work, we did not enable loop nest transformation in our experiment. Studying the interaction between our technique and other locality-enhancing techniques like *loop fusion*, *loop fission*, *loop permutation* and *loop tiling* is our future work. However, optimizations applied on loop bodies, like strength reduction, induction variable elimination and cross-iteration common subexpression elimination that do not change the loop nest structure, are still invoked.

We implemented the algorithm for computing the reference window size given in Gannon et al's paper [10] which is used in the 0/1 knapsack problem. We have also implemented the knapsack problem formulation (i.e., generating the constraints) and a greedy algorithm to get the optimal solution for it in the MIPSpro compiler. A dynamic-programming algorithm for general 0/1 knapsack problem is interesting but was not required since in the benchmarks we evaluated perfect loop nests dominate. We did not consider scalar references for `nt` hint, as scalar variables only consume a small portion of cache space.

SimpleScalar uses MIPS instruction set with a few minor differences. Each instruction word in SimpleScalar is of length 64 bits, of which the most significant 16 bits are not used at present. This 16-bit field is called *annotation* field in SimpleScalar, which is used by us to carry cache hint in our experiment. During code generation, memory instructions are given `nt` hint according to the solution of the 0/1 knapsack problem. In response to this modification on ISA, we modified the simulation mechanism as well. We implemented the modified LRU algorithm which does not change the rank a the cache line for accesses with `nt` hint.

We chose two representative loop kernels, **mxm**, in which most data accesses are column-major, and **vpenta**, in which most data accesses are row-major. Both of them are from SPEC92 `093.nasa7` benchmark written in Fortran. Besides, to evaluate the effectiveness of our approach on large benchmarks, we also included one complete benchmark, `tomcatv` from SPEC95 with `train` data set, in our workload. We experimented our approach on caches of varying sizes (ranging from 4K bytes to 32K bytes) and varying associativity (2-way and 4-way). Note that for direct-mapped cache, the replacement algorithm and cache hint do not play any role. In our experimental work, we used a fixed cache line size of 16 bytes.

5.2 Performance Results

The cache miss rates of the conventional cache and that of `nt` hint assisted cache are compared in Table 1. The cache miss results of these two types of cache are obtained by running exactly the same program generated by our compiler on the SimpleScalar simulator (simulating, respectively, the LRU replacement algorithm and the modified LRU replacement algorithm).

Table 1. Effectiveness of our approach in reducing cache misses. Column "LRU" reports cache miss rates of conventional caches with LRU replacement. Column "LRU+hint" report cache miss rates `nt` hint assisted caches which uses a modifies LRU replacement. Column "Red" gives the percentage reduction in cache miss rates due to our approach

Benchmark	Cache Size											
	4K			8K			16K			32K		
	LRU	LRU + hint	Red.	LRU	LRU + hint	Red.	LRU	LRU + hint	Red.	LRU	LRU + hint	Red.
Result On 4-way Associative Caches												
`mxm`	35%	29.4%	16%	35%	15%	57.1%	8%	8%	0%	8%	8%	0%
`vpenta`	21.7%	21.3%	1.8%	21.6%	19.7%	8.8%	17.2%	13.5%	21.5%	13.2%	10.9%	17.4%
`tomcatv`	21%	21.5%	-2.4%	20.9%	18.5%	11.5%	20.1%	17.1%	14.9%	16%	14%	12.5%
average			5.1%			25.8%			12.1%			10%
Result On 2-way Associative Caches												
`mxm`	35%	28.7%	18%	21.9%	15.2%	30.6%	8.1%	8%	1.2%	4%	4%	0%
`vpenta`	21.9%	22.18%	-1.3%	21.7%	18.9%	12.9%	19.5%	16.3%	16.4%	18.6%	16.2%	12.9%
`tomcatv`	22.9%	23%	-0.4%	22.5%	22.6%	-0.4%	20%	18.1%	9.5%	16.9%	14.8%	12.4%
average			5.4%			14.3%			9%			8.4%

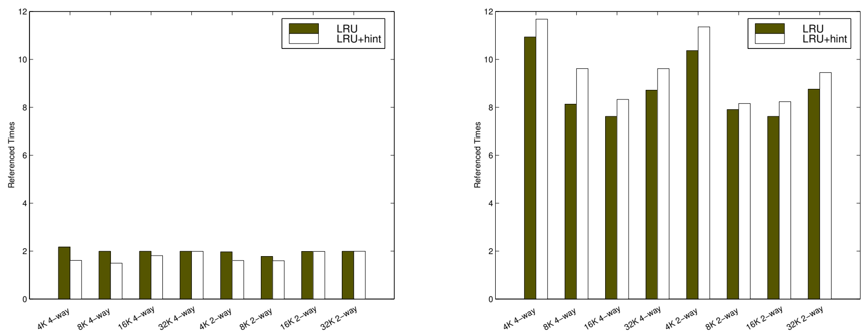
Our approach shows most performance benefits on **mxm** for 8K byte 4-way cache. It reduces the cache miss rate from 35% to 15% (a 57.1% reduction on the number of cache misses). As elaborated in Section 3, the key to achieve satisfactory overall cache locality is to ensure that reuse of array references of **A** is materialized, since in this example, reference of **C** has distant reuse and references of **B** are loop-invariant. But, in a conventional cache, cache pollution caused by array **C** prevents array **A** from enjoying its temporal locality, leading to poor locality on a cache of size 4K and 8K bytes. For 8K byte cache, 41.3% of the executed memory instructions are given the `nt` cache hint by our approach. This ensures that data accessed by normal memory instructions (references of **A** in this case) stay in the cache for a relatively longer time which in turn results in better temporal locality.

For 2-way 8K byte cache, our approach is also quite effective, reducing the number of cache misses in **mxm** by 30.6%. The percentage reduction achieved on a 2-way cache is lower than that achieved by a 4-way cache. Although this is counter-intuitive, we observe that, even for the conventional cache with the original LRU replacement, **mxm** achieves lower cache miss rates on a 2-way 8K byte cache than on a 4-way cache of the same size. This could be due to higher conflict misses as 4-way associativity results in fewer sets (128 sets) than 2-way associativity (256 sets) on a 8K byte cache.

Our approach performs consistently better over conventional cache for larger cache sizes (16K and 32K bytes). For caches of relatively smaller sizes (especially 4K bytes), our approach performs marginally better than the conventional cache, but not consistently. The reason for this is that when data accessed by an instruction with `nt` hint is brought in, its life time in the cache is typically much shorter than that in a conventional cache. Although this is beneficial for other data with temporal locality, the short cache life-time of the accessed data jeopardizes spatial locality since it may be replaced before the adjacent data items are accessed. On a cache of small size, this happens more frequently.

To verify the above conjecture, we designed an experiment in which each cache object is classified as a *regular* object or an *nt-hint* object depending on whether the data object accessed is brought into the cache using a regular memory instruction or with an `nt` hint memory instruction. We measured the number of references for each cache block during its life-time (from the time the cache block is brought in to the time it is replaced). Using this we compute the average number of references for regular objects and `nt`-hint objects. We compute these values for both classes of objects with the original as well as the modified LRU replacement algorithm. Note, in all our experiments the code run in the simulator is the same (one which includes `nt`-hint memory instruction). Only the replacement policy used (original LRU or modified LRU) is different for the different caches.

Figure 5(a) shows the average number of references for `nt`-hint objects for `tomcatv` benchmark. It can be seen that the average number of references remain the same between the original and the modified LRU replacement for 32K byte cache. However, for small cache sizes, there is a decrease in the average number of references. This shows that spatial locality exploited in `nt`-hint objects is lower in `nt`-hint assisted caches, especially when the cache size is smaller. In other words, the locality of the `nt`-hint objects is really sacrificed. For reference purposes, we also show the average number of references for regular objects in Figure 5(b). It can be seen that the modified LRU algorithm (with `nt` hint) improves the locality of regular objects in all cache sizes. These two graphs (refer to Figure 5) tell us the key to reduce the cache miss ratio on the studied architectures is to avoid/minimize the degradation of the locality exploited in `nt`-hint objects while enhancing the locality of exploited in regular objects. Fortunately, for most cases the benefits achieved in temporal locality exploited in regular objects by our approach dominate the possible loss on spatial locality exploited in `nt`-hint objects. This is evidenced by the positive average reduction on cache misses we achieved for all cache sizes we considered.



(a) Average Number of References for `nt`-hint Objects

(b) Average Number of References for Regular Objects

Fig. 5. Impact of our approach on locality of regular and `nt`-hint objects.

We observe that our approach is more effective on caches of higher associativity. As shown, our approach reduces the cache miss rate by a larger extent for 4-way associative caches than for 2-way associative caches. One possible reason for this is that our problem formulation does not take cache conflicts into account. In our problem formulation given in Section 4, we optimistically assumed that the residency of reference windows is only constrained by the effective cache size. This assumption gives us a simple problem formulation; but it suffers from not considering conflict misses which is non-negligible on caches of low associativity. Our future work will consider using conflict-avoiding techniques like data padding to improve the effectiveness of our approach.

Next we report the impact of reduced cache misses (due to `nt` hint) on program performance. For this, we obtain program execution time, expressed in execution cycles, from SimpleScalar simulator. We simulate a superscalar processor which issues 2 instructions in a clock cycle and employs out-of-order instruction issue and out-of-order execution. We consider one level of cache: I-cache of 16K bytes, and the size of D-cache varies between 4K and 32K bytes. The cache hit latency is 1 cycle, and the cache miss penalty is 40 cycles. Performance results for a conventional cache and a cache with `nt` hints are reported in Table 2.

We observe that the reduction in cache misses (due to `nt` hints) does result in a corresponding reduction in the execution time, although not exactly by the same/similar amount. This is because cache miss rate is not the only factor affecting program performance, especially in out-of-order issue processors. In general, we observe that the

Table 2. Effectiveness of our approach in improving program performance. This table shows the normalized execution time of benchmark programs running on a conventional cache with LRU replacement (shown in column "LRU") and on a cache with modified LRU replacement (shown in column "LRU+hint"). For each program, execution time shown is normalized using the execution time of the program on a conventional 4K byte, 4-way associative D-cache

Benchmark	4K			8K			16K			32K		
	LRU	LRU + hint	Red.	LRU	LRU + hint	Red.	LRU	LRU + hint	Red.	LRU	LRU + hint	Red.
Result On 4-way Associative Caches												
`mxm`	1	0.76	24%	1	0.73	27%	0.58	0.58	0%	0.58	0.57	1.7%
`vpenta`	1	1.04	-4%	1	1	0%	0.92	0.85	7.6%	0.90	0.78	13.3%
`tomcatv`	1	1.03	-3%	1	0.94	6%	0.98	0.91	7.1%	0.89	0.82	7.9%
average			5.7%			11%			4.9%			7.6%
Result On 2-way Associative Caches												
`mxm`	1	0.74	26%	0.73	0.72	0.3%	0.58	0.58	0%	0.44	0.44	0%
`vpenta`	1	1.04	-4%	1	0.98	2%	0.98	0.93	5.1%	0.96	0.93	3.1%
`tomcatv`	1.02	1.07	-4.9%	1.01	1.07	-5.9%	0.98	0.92	6.1%	0.93	0.85	8.6%
average			5.7%			-1.2%			3.7%			3.9%

cache miss rate reduction achieved by our approach is accompanied by a corresponding performance improvement. With the widening speed gap between processor and memory, our approach can have more performance impact on future microprocessors.

6 Related Work

Improving cache performance has attracted a lot of attention from both the architecture and compiler perspective. Specifically, enhancing instruction set with cache hints is pioneered by Chi and Dietz. They studied an architecture innovation by introducing cache-bypassing memory instructions [6, 7]. In their architecture model, data accessed by cache-bypassing memory instructions is not allocated a cache line. Their approach is helpful to avoid cache pollution, but severely compromises spatial locality. By using cache hints, we can get better temporal locality without sacrificing the spatial locality significantly.

Wang et al studied a hypothetical architecture similar to the one considered in this paper [19], and proposed a heuristic compiler algorithm for this architecture. However our work differs from their work in two major aspects: (i) we performed an in-depth study on the compiler algorithm while they focused on the architectural implementation; (ii) we presented a systematic formulation while they used an ad-hoc algorithm. Lastly, their algorithm does have the cyclic dependency problem mentioned in Section 1. In a future work, we plan to compare our approach with their heuristic method.

Anantharaman and Pande studied the problem of optimizing loop execution on embedded systems with scratch-pad memory and without cache [2]. Interestingly, they formulated the problem as a 0/1 knapsack problem as well. However, the problem they studied is different from ours since scratch-pad memory differs from the cache in that it is free of hardware interference.

7 Conclusions

Improving cache performance is of significant importance in modern processors. In this paper we exploited compiler-assisted cache management which utilizes the cache more efficiently to achieve better performance. In particular, we studied the problem of determining the subset of references that should be given `nt` (stands for "non-temporal") cache hints to minimize the cache miss rate. We observe the relationship between cache miss rate and cache-residency of reference windows in Section 3. This observation forms the basis for our formulation that in order for an array reference to realize its temporal reuse, its reference window must be fully accommodated in the cache. We then formulated the problem as a 0/1 knapsack problem for the following two cases: (i) only self-reuse exists, and (ii) group-reuse exists. To the best of our knowledge this is the first systematic formulation of this problem. We evaluated our approach by implementing it in a re-engineered MIPSpro compiler generating SimpleScalar instructions and running it through SimpleScalar simulator. Our simulation results show that our approach exploited the architecture potential well. It reduced the number of data cache misses by up to 57%, and program execution time by up to 25.7%. Our plan for future work includes performing a comprehensive evaluation on the sensitivity of our approach to cache associativity and cache line size, integrating our approach with other locality-enhancing techniques, and comparing it with related work.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.
2. S. Anantharaman and S. Pande. Compiler optimization for real time execution of loops on limited memory embedded systems. In *Proceedings of 1998 IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec 1998.
3. Doug Burger and Todd Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, Univ of Wisconsin, 1997.
4. Douglas C. Burger, James R. Goodman, and Alain Kägi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report WMADISONCS CS-TR-95-1261, University of Wisconsin-Madison, Computer Sciences Department, 1995.
5. David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proc. of SIGPLAN PLDI '90*, pages 53–65, White Plains, N. Y., Jun. 1990.
6. C.-H. Chi and H. Dietz. Improving cache performance by selective cache bypass. In *Twenty-Second Annual Hawaii International Conference on System Sciences*, pages 277–285, 1989.
7. Chi-Hung Chi and Hank Dietz. Unified management of registers and cache using liveness and cache bypass. In *Proc. of SIGPLAN PLDI '89*, pages 344–355, Portland, Ore., Jun. 1989.
8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 1992.
9. Intel Corp. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, Jun 2002.

10. Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global programming transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.
11. Guang R. Gao, Vivek Sarkar, and Shaohua Han. Locality analysis for distributed shared-memory multiprocesors. In *Proc. of the 1996 International Workshop on Languages and Compilers for Parallel Computing(LCPC)*, San Jose, California, Aug 1996.
12. Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Conf. Proc., 1997 Intl. Conf. on Supercomputing*, pages 317–324, Vienna, Austria, Jul. 1997.
13. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., Inc., San Francisco, 2nd edition, 1996.
14. J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, 5(4):438–456, August 1988.
15. Kathryn S. McKinley and Olivier Temam. Quantifying loop nest locality using spec'95 and the perfect benchmarks. *ACM Transactions on Computer Systems (TOCS)*, 17(4):288–336, 1999.
16. T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.
17. P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.
18. Vivek Sarkar and Nimrod Megdido. An analytical model for loop tiling and its solution. In *Proceedings of IEEE 2000 International Symposium on Performance Analysis of Systems and Software*, Austin, TX, Apr 2000.
19. Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the 11th International Conference on Parallel Architecture and Compilation Techniques(PACT'02)*, Charlottesville, Virginia, Sept 2002.
20. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. of SIGPLAN PLDI '91*, pages 30–44, Toronto, Ont., Jun. 1991.
21. Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proc. of MICRO-29*, pages 274–286, Paris, Dec. 1996.
22. Hongbo Yang, R. Govindarajan, Guang R. Gao, and Ziang Hu. A problem formulation of assisting cache replacement by compiler. Technical Report 47, Computer Architecture and Parallel Systems Laboratory, University of Delaware, 2003.