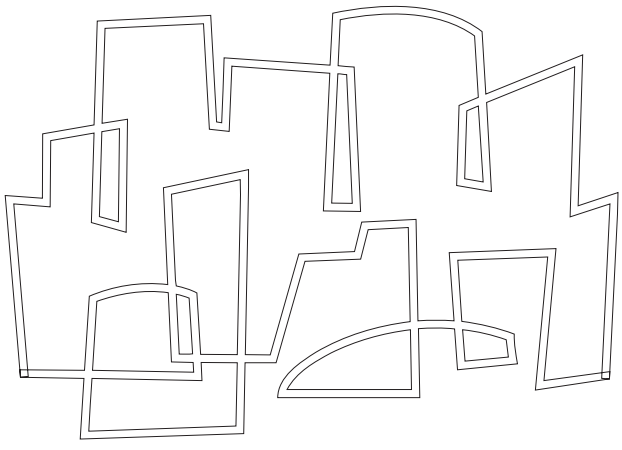
A Code Reordering Transformation for Improved Cache Performance

Prabhat Jain, Srinivas Devadas

2001, March

Computation Structures Group
Memo 436



A Code Reordering Transformation for Improved Cache Performance

Prabhat Jain, Srinivas Devadas
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
e-mail: {prabhat,devadas}@mit.edu

Abstract

We prove theorems that show that if we can reorder a program's memory reference stream such that the reordered memory reference stream satisfies a disjointness property, then the transformed program corresponding to the reordered stream is guaranteed to have fewer misses for any cache with arbitrary size or organization so long as the cache uses the LRU replacement policy. We can apply these results to reorder instructions within a basic block, to transform loops, to reorder blocks within a procedure, or to reorder procedures within a program so as to improve hit rate for any cache that uses LRU replacement.

Based on these theorems, we develop algorithmic methods for program transformation to improve cache performance. While there has been a lot of work in improving cache performance using program transformations, our work differs from previous work in that it can be applied at many different levels, from the basic block to procedural levels, and does not require fixing the size or organization of a cache prior to program transformation.

We present preliminary experimental results that show that reordering based on our methods can result in significant improvements in hit rate and program performance.

1 Introduction

Caches are ubiquitous in modern processors and with processor speeds increasing faster than memory access speeds, cache hit rate is becoming more significant in determining overall application performance [5]. Optimizing programs to improve cache performance, measured by hit rate, is thus an important avenue of research. Caches vary greatly in size and organization, and multilevel memory hierarchies with vastly different sizes of caches are common in modern computer systems. While there has been quite some work in optimizing program performance for specific processors and memory systems, general techniques for program optimization that improve cache performance regardless of cache size or organization have the attractive features of portability, flexibility, and backward and forward compatibility.

In this paper, we provide a theoretical basis for improving the cache performance of programs by code reordering and transformation, which results in reordering the memory reference stream of the program. We prove theorems that show that if we can reorder a program's memory reference stream such that the reordered memory reference stream satisfies a

disjointness property, then the transformed program corresponding to the reordered stream is guaranteed to have fewer misses for any cache with arbitrary size or organization so long as the cache uses the LRU replacement policy. To elaborate, if a memory reference stream R_{12} can be reordered into a concatenation of two memory reference streams R_1 and R_2 such that R_1 and R_2 are disjoint streams, i.e., no memory address in R_1 is in R_2 and vice versa, then the number of misses produced by R_{12} for *any* cache with LRU replacement is guaranteed to be greater than or equal to the number of misses produced by the stream $R_1@R_2$, where @ denotes concatenation. Thus, reordering to produce disjoint memory reference streams is guaranteed to improve performance, as measured by cache hit rate.

We can apply these theoretical results to reorder instructions within a basic block, to transform loops, to reorder blocks within a procedure, or to reorder procedures within a program so as to improve hit rate for any cache that uses LRU replacement.

Based on these results, we develop algorithmic methods for program transformation to improve cache performance. While there has been a lot of work in improving cache performance using program transformations, our work differs from previous work in that it can be applied at many different levels, from the basic block to procedural levels, and does not require fixing the size or organization of a cache prior to program transformation. Work on cache oblivious algorithms [2, 10] are similar in that these algorithms apply to caches of all sizes, but these algorithms assume optimal replacement, which is not implementable in practice, and further these algorithms have thus far only been applied to specific programs such as FFT. We present preliminary experimental results that show that reordering based on our methods can result in significant improvements in hit rate and program performance.

The paper is organized as follows. In Section 2 we describe related work and give an outline of our approach. In Section 3 we prove some theorems that form the basis of our reordering program transformations. In Section 4 we discuss the program transformations we consider in this paper based on the theorems. In Section 5 we describe the algorithm based on the theorems which uses a set of transformations to transform the code. We also provide a metric to approximate the application of the theorems. In Section 6 we give experimental results that show that we can achieve significant improvement based on the reordering transformations in some cases. In Section 7 we conclude the paper with some directions for future work.

2 Related Work and Our Approach

There has been lot of work done that involves loop transformation based optimizations to improve data locality. Some examples of the loop transformations are loop fusion, loop distribution, loop permutation, and loop reversal. In [13], an approach is proposed to combine various loop transformations. Their approach uses a model that estimates the total machine cycle time taking into account cache misses, software pipelining, register pressure, and loop overhead. In [8] compiler optimizations are presented based on a cost model. The model computes both temporal and spatial reuse of cache lines to find desirable loop organizations. This cost model is used to apply the compound loop transformations consisting of loop permutation, loop fusion, loop distribution, and loop reversal. In [3], reuse analysis is suggested for some loop and program transformations. In [12], an algorithm is presented

that improves the locality of a loop nest by transforming the code via interchange, reversal, skewing, and tiling. It uses a formulation of reuse and locality and a loop transformation theory to unify the various transforms as unimodular matrix transformations. In [4] a cache miss analysis framework called Cache Miss Equations(CME) is presented. It expresses the memory reference and cache conflict behavior in terms of sets of equations. Their approach uses the CMEs to apply padding and tiling transformations. In [1] a tile selection algorithm is presented that aims at eliminating self-interference and simultaneously minimizing capacity and cross-interference misses. In [9] an approach is presented that uses data and computation reordering to improve memory hierarchy utilization for irregular applications on systems with multi-level memory hierarchies. In [11] some program transformations are presented to enable tiling for a class of nontrivial imperfectly-nested loops such that the cache locality is improved. In [7] blocked algorithms are studied in the context of the cache parameters and their impact on the cache interference. In [6] an approach that combines loop transformations and layout optimizations in an integrated framework is presented. The loop transformations are used to improve temporal locality and the data layout optimizations are used to improve spatial locality.

Work on cache oblivious algorithms [2, 10] are similar to our work in that these algorithms apply to caches of all sizes, but these algorithms assume optimal replacement, which is not implementable in practice, and further these algorithms have thus far only been applied to specific programs such as FFT.

Our work is not limited to loop transformations discussed above and we provide a theoretical basis to apply the reordering transformations at different levels of the program. Our approach is to reorder the memory references to satisfy the disjointness property whenever possible under the data dependency constraints and use a disjointness metric when the disjointness property cannot be satisfied completely. Our work differs from the cache oblivious algorithms because we use cache oblivious program transformations rather than cache oblivious algorithms. Moreover, we assume the use of the LRU replacement policy rather than the optimal replacement policy.

3 Disjoint Sequence Merge Theorems

We present the theoretical results that form the basis of the code reordering program transformations.

3.1 Definitions

Let R_1 and R_2 be two disjoint reference sequences merged without re-ordering to form a sequence S. That is, S is formed by interspersing elements of R_2 within R_1 (or vice versa) such that the order of elements of R_1 and R_2 within S is unchanged. For example, consider the sequence $R_1 = a, b, a, c$ and $R_2 = d, d, e, f$. These sequences are disjoint, i.e., they do not share any memory reference. The sequence S can be a, d, d, b, e, a, c, f but not a, d, d, a, b, c, e, f because the latter has reordered the elements of R_1.

Let $R_1^{'}$ be the R_1 sequence padded with the null element ϕ in the position where R_2 elements occur in S such that $\parallel S \parallel = \parallel R_1^{'} \parallel = N$. If $S = a, d, d, b, e, a, c, f$ then $R_1^{'} =$

$a, \phi, \phi, b, \phi, a, c, \phi$. So, based on the above description, if $S(x) \in R_1$ then $S(x) = R_1^{'}(x)$. We define the null element ϕ to always result in a hit in the cache.

Let $C(S,t)$ be the cache state at time t for the sequence S. Let $C(R_1^{'},t)$ be the cache state at time t for the sequence $R_1^{'}$. There are no duplicate elements in a cache state. Let the relation $X \preceq Y$ indicate that $X \subseteq Y$ and the order of the elements of X in Y is same as the order in X. For example, if $X = \{a, b, e\}$ and $Y = \{f, a, g, b, h, e\}$ then $X \preceq Y$ because $X \subseteq Y$ and the order of a, b, and e in Y is the same as the order in X.

For a direct-mapped cache C is an array; for a fully associative cache with an LRU policy C is an ordered set; and for a set-associative cache with the LRU policy C is an array of ordered sets.

3.2 Disjoint Sequence Merge Theorem

Theorem 1: Given a cache C with an organization (direct mapped, fully-associative or set-associative) and the LRU replacement policy, and two disjoint reference sequences R_1 and R_2 merged without re-ordering to form a sequence S. The number of misses m_1 resulting from applying the sequence R_1 to $C \leq$ the number of misses M resulting from applying the sequence S to C.

Sketch of the proof: Let $m_1^{'}$ be the number of misses of R_1 in S. Let $m_2^{'}$ be the number of misses of R_2 in S. So, $M = m_1^{'} + m_2^{'}$ is the total number of misses in S. For a direct-mapped cache we show by induction that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'},t)$. For a fully-associative cache we show by induction that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \preceq C(R_1^{'},t)$. For a set-associative cache we show by induction that for any time t, $0 \leq t \leq N$, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'},t)[i]$, where p is the number of ordered sets in the set-associative cache. This implies that if an element of R_1 results in a miss for the sequence $R_1^{'}$ it would result in a miss in the sequence S. So, we have $m_1 \leq m_1^{'}$ and by symmetry $m_2 \leq m_2^{'}$. So, $m_1 + m_2 \leq m_1^{'} + m_2^{'}$ or $m_1 + m_2 \leq M$. Thus we have $m_1 \leq M$ and $m_2 \leq M$.

3.2.1 Direct Mapped Cache

We show that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'},t)$. Every element e maps to an index $Ind(e)$ that can be used to lookup the element in the cache state C.

For $t = 0$, $C(S,0) \cap R_1 \subseteq C(R_1^{'},0)$.
Assume for time t, $C(S,t) \cap R_1 \subseteq C(R_1^{'},t)$.
For time $t+1$, let $f = S(t+1)$ and let $i = Ind(f)$ and let $e = C(S,t)[i]$ and let $x = C(R_1^{'},t)[i]$.

Case 0 (hit): The element f results in a hit in $C(S,t)$. So, $e \equiv f$ and $C(S,t+1) = C(S,t)$. If $f \in R_1$, $R_1^{'}(t+1) \equiv f$ and from the assumption at time t, $C(R_1^{'},t)[i] \equiv f$. Since the element f results in a hit in $C(R_1^{'},t)$, $C(R_1^{'},t+1) = C(R_1^{'},t)$. So, $C(S,t+1) \cap R_1 \subseteq C(R_1^{'},t+1)$. If $f \in R_2$, then $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'},t+1) = C(R_1^{'},t)$. So, $C(S,t+1) \cap R_1 \subseteq C(R_1^{'},t+1)$.

Case 1 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_2$ and $e \in R_1$. The new cache state for the sequence S is $C(S,t+1) = C(S,t) - \{e\} \cup \{f\}$. This implies that $C(S,t+1) \cap R_1 \subseteq C(S,t) \cap R_1$. The element $R_1^{'}(t+1) \equiv \phi$ so $C(R_1^{'},t+1) = C(R_1^{'},t)$. So,

$C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 2 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_2$ and $e \in R_2$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. This implies that $C(S, t+1) \cap R_1 \equiv C(S,t) \cap R_1$. The element $R_1^{'}(t+1) \equiv \phi$ so $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 3 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_1$ and $e \in R_1$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. The element $R_1^{'}(t+1) \equiv f$ from the construction of S and $R_1^{'}$. The new cache state for the sequence $R_1^{'}$ is $C(R_1^{'}, t+1) = C(R_1^{'}, t) - \{x\} \cup \{f\}$. From the assumption at time t $x \equiv e$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

Case 4 (miss): The element f results in a miss in $C(S,t)$ and $f \in R_1$ and $e \in R_2$. The new cache state for the sequence S is $C(S, t+1) = C(S,t) - \{e\} \cup \{f\}$. The element $R_1^{'}(t+1) \equiv f$ from the construction of S and $R_1^{'}$. The new cache state for the sequence $R_1^{'}$ is $C(R_1^{'}, t+1) = C(R_1^{'}, t) - \{x\} \cup \{f\}$. Since $e \in R_2$ and the assumption at time t, $(C(S,t) - \{e\}) \cap R_1 \subseteq (C(R_1^{'}, t) - \{x\})$. So, $C(S, t+1) \cap R_1 \subseteq C(R_1^{'}, t+1)$.

3.2.2 Fully-associative Cache

Lemma 1: Let $C(S,t) = \{L, y\}$, where L is an ordered subset of elements and y is the LRU element of $C(S,t)$. Let $C(R_1^{'}, t) = \{M, z\}$, where M is an ordered subset of elements and z is the LRU element of $C(R_1^{'}, t)$. If $y \in R_2$, then $z \notin L$.
Proof: Based on the construction of $R_1^{'}$ and S, if M is not null, the last reference of z in both the sequences should occur at the same time $t_1 < t$ and between t_1 and t the number of distinct elements in $R_1^{'} \leq$ the number of distinct elements in S. If M is null, then there is only one element in C at time t because we have referenced z over and over, so t can be anything but $t_1 = t$ and the number of distinct elements following z in $R_1^{'} = 0$ as in S. Let $\parallel C \parallel = c$. For $C(R_1^{'}, t) = \{M, z\}$, let the number of distinct elements following z be n. Since z is the LRU element in $C(R_1^{'}, t)$, $n = c - 1$. Let us assume that $z \in L$. Let $L = \{L_1, z, L_2\}$ and $\parallel L_1 \parallel = l_1$, $\parallel L_2 \parallel = l_2$, $\parallel L \parallel = c - 1$. For $C(S,t) = \{L, y\} = \{L_1, z, L_2, y\}$, let the number of distinct elements following z be m and $m = l_1$. So, $m < c - 1$. So, $m < n$ and that contradicts the assertion on the number of distinct elements. Therefore, $z \notin L$.

We show that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \preceq C(R_1^{'}, t)$.

For $t = 0$, $C(S,0) \cap R_1 \preceq C(R_1^{'}, 0)$.
Assume for time t, $C(S,t) \cap R_1 \preceq C(R_1^{'}, t)$.
For time $t+1$, let $x = S(t+1)$.

Case 0 (hit): $x \in R_1$ and x results in a hit in $C(S,t)$. Let $C(S,t) = \{L_1, x, L_2\}$, where L_1 and L_2 are subsets of ordered elements. The new state for the sequence S is $C(S, t+1) = \{x, L_1, L_2\}$. Let $C(R_1^{'}, t) = \{M_1, x, M_2\}$, where M_1 and M_2 are subsets of ordered elements. $C(R_1^{'}, t+1) = \{x, M_1, M_2\}$. From the assumption at time t, $\{L_1, x, L_2\} \cap R_1 \preceq \{M_1, x, M_2\}$. So, $\{L_1\} \cap R_1 \preceq \{M_1\}$ and $\{L_2\} \cap R_1 \preceq \{M_2\}$. Thus $\{L_1, L_2\} \cap R_1 \preceq \{M_1, M_2\}$. So,

$C(S, t+1) \cap R_1 \preceq C(R_1^{'}, t+1)$.

Case 1 (hit): $x \in R_2$ and x results in a hit in $C(S,t)$. Let $C(S,t) = \{L_1, x, L_2\}$, where L_1 and L_2 are subsets of ordered elements. The new state for the sequence S is $C(S, t+1) = \{x, L_1, L_2\}$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. Since $x \in R_2$, $C(S, t+1) \cap R_1 \equiv C(S,t) \cap R_1$. So, $C(S, t+1) \cap R_1 \preceq C(R_1^{'}, t+1)$.

Case 2 (miss): $x \in R_2$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_2$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S, t+1) = \{x, L\}$. Since $x \in R_2$ and $y \in R_2$, $C(S, t+1) \cap R_1 \equiv C(S,t) \cap R_1$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So, $C(S, t+1) \cap R_1 \preceq C(R_1^{'}, t+1)$.

Case 3 (miss): $x \in R_2$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_1$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S, t+1) = \{x, L\}$. Since $x \in R_2$ and $y \in R_1$, $C(S, t+1) \cap R_1 \preceq C(S,t) \cap R_1$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv \phi$ and $C(R_1^{'}, t+1) = C(R_1^{'}, t)$. So, $C(S, t+1) \cap R_1 \preceq C(R_1^{'}, t+1)$.

Case 4 (miss): $x \in R_1$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_2$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S, t+1) = \{x, L\}$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv x$. Let $C(R_1^{'}, t) = \{M, z\}$, where M is a subset of ordered elements. The new state for the sequence $R_1^{'}$ is $C(R_1^{'}, t+1) = \{x, M\}$. From Lemma 1, $z \notin L$. So, $\{x, L\} \cap R_1 \preceq \{x, M\}$. So, $C(S, t+1) \cap R_1 \preceq C(R_1^{'}, t+1)$.

Case 5 (miss): $x \in R_1$ and x results in a miss in $C(S,t)$ and the LRU element $y \in R_1$. Let $C(S,t) = \{L, y\}$, where L is a subset of ordered elements and y is the LRU element of $C(S,t)$. The new state for the sequence S is $C(S, t+1) = \{x, L\}$. From the construction of $R_1^{'}$, $R_1^{'}(t+1) \equiv x$. Since $y \in R_1$, the LRU element of $C(R_1^{'}, t)$ is also y due to the assumption at time t. Let $C(R_1^{'}, t) = \{M, y\}$, where M is a subset of ordered elements. The new state for the sequence $R_1^{'}$ is $C(R_1^{'}, t+1) = \{x, M\}$. From the assumption at time t, $\{L\} \cap R_1 \preceq \{M\}$. Thus, $\{x, L\} \cap R_1 \preceq \{x, M\}$. So, $C(S, t+1) \cap R_1 \preceq C(R_1^{'}, t+1)$.

3.2.3 Set-associative Cache

Assume that there are p ordered sets in C that can be referred as $C[0], ..., C[p-1]$. Every element e maps to an index $Ind(e)$ such that $0 \leq Ind(e) \leq p-1$ that is used to lookup the element in the ordered set $C[Ind(e)]$. We show for $0 \leq t \leq N$, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'}, t)[i]$.

For $t = 0$, $0 \leq i \leq p-1$, $C(S,0)[i] \cap R_1 \preceq C(R_1^{'}, 0)[i]$.
Assume for time t, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'}, t)[i]$.
For time $t+1$, let $x = S(t+1)$, and $j = Ind(x)$.
For $i \neq j$ ($0 \leq i \leq j-1$ and $j+1 \leq i \leq p-1$), $C(S, t+1)[i+1] = C(S,t)[i+1]$ and $C(R_1^{'}, t+1)[i+1] = C(R_1^{'}, t)[i+1]$ because the element x does not map in the ordered set $C[i]$. So, $C(S, t+1)[i] \cap R_1 \preceq C(R_1^{'}, t+1)[i]$.

For $i = j$ using the assumption at time t and the proof for the fully-associative cache we have $C(S, t+1)[i] \cap R_1 \preceq C(R_1^{'}, t+1)[i]$.
Therefore, $C(S, t+1)[i] \cap R_1 \preceq C(R_1^{'}, t+1)[i]$ for $0 \leq i \leq p-1$.

3.3 Concatenation Theorem

Theorem 2: Given two disjoint reference sequences R_1 and R_2 merged in any arbitrary manner without re-ordering the elements of R_1 and R_2 to form a sequence S and given the concatenation of R_1 and R_2 indicated by $R_1@R_2$, the number of misses produced by S for *any* cache with the LRU replacement is greater than or equal to the number of misses produced by $R_1@R_2$.
Proof: Let m_1 indicate the number of misses produced by the sequence R_1 alone and m_2 indicate the number of misses produced by the sequence R_2 alone. The number of misses produced by $R_1@R_2$ is $m_1 + m_2$. Let M be the number of misses produced by S. We show that $m_1 + m_2 \leq M$. Let $m_1^{'}$ indicate the number of misses of R_1 in S and let $m_2^{'}$ indicate the number of misses of R_2 in S. For a direct-mapped cache we showed that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \subseteq C(R_1^{'}, t)$ and for a fully-associative cache we showed that for any time t, $0 \leq t \leq N$, $C(S,t) \cap R_1 \preceq C(R_1^{'}, t)$. For a set-associative cache we showed by induction that for any time t, $0 \leq t \leq N$, $0 \leq i \leq p-1$, $C(S,t)[i] \cap R_1 \preceq C(R_1^{'}, t)[i]$, where p is the number of ordered sets in the set-associative cache. This implies that if an element of R_1 results in a miss for the sequence $R_1^{'}$ it would result in a miss in the sequence S. Thus, $m_1 \leq m_1^{'}$. By symmetry, we have $m_2 \leq m_2^{'}$. So, $m_1 + m_2 \leq m_1^{'} + m_2^{'}$ or $m_1 + m_2 \leq M$, where $M = m_1^{'} + m_2^{'}$ is the total number of misses in S. Therefore, the number of misses M produced by S is greater than or equal to the number of misses $m_1 + m_2$ produced by $R_1@R_2$.

4 Program Transformations

We consider code reordering transformations that are based on the theorems described in Section 3. We show three examples of transformations: a loop, a loop with a conditional, and a dependency graph scheduling based transformation.

A loop transformation is shown in Figure 1. In the original code shown in Figure 1(a) accesses the three arrays `A`, `B`, and `C` in an interleaved manner. The transformed code is shown in Figure 1(b). The reordering transformation in this case separates the accesses to the three arrays but maintains the order of accesses within the individual arrays. In the transformed code the accesses to `A`, `B`, and `C` satisfy the disjointness property discussed in Section 3. Therefore, the transformed code would incur less misses for the three array accesses.

A transformation for a loop with a conditional is shown in Figure 2. In the original code shown in Figure 2(a) the accesses to the arrays `A` and `B` are interleaved. The transformed code is shown in Figure 2(b). The code reordering transformation in this case separates the accesses to `A` and `B` based on the conditional and the use of modified loops but the transformation maintains the order of accesses within the individual arrays `A` and `B`. In the transformed code the accesses to `A` and `B` satisfy the disjointness property discussed

```
for(i=0; i < N; i++) {
  A[i] = 0;
  B[i] = 0;
  C[i] = 0;
}
```

(a)

```
for(i=0; i < N; i++) {
  A[i] = 0;
}
for(i=0; i < N; i++) {
  B[i] = 0;
}
for(i=0; i < N; i++) {
  C[i] = 0;
}
```

(b)

Figure 1: (a) Loop Code, (b) Transformed Loop Code

```
for(i=0; i < N; i++) {
  if (i%2 == 0)
    A[i] = x;
  else
    B[i] = y;
}
```

(a)

```
for(i=0; i < N; i+=2) {
  A[i] = x;
}
for(i=1; i < N; i+=2) {
  B[i] = y;
}
```

(b)

Figure 2: (a) Loop Code with Conditional, (b) Transformed Code

in Section 3. Therefore, the transformed code would incur less misses for the three array accesses.

When there are dependency constraints that a code reordering transformation needs to satisfy, we represent the dependence constraints as a graph where the nodes can represent references, basic blocks, loops, and procedures. We consider a scheduling transformation for a dependency graph and an example is shown in Figure 3. In Figure 3(a) a dependency graph G is shown, where $D_i \rightarrow D_j$ means that D_i has to be performed before D_j. Based on the dependency constraints, the two possible schedules are shown in Figure 3(b) and Figure 3(c). If R_{12} is the set of accesses corresponding to $\{D_1, D_2\}$, R_{34} is the set of accesses corresponding to $\{D_3, D_4\}$, R_{13} is the set of accesses corresponding to $\{D_1, D_3\}$, and R_{24} is the set of accesses corresponding to $\{D_2, D_4\}$ then the schedule in Figure 3(b) is guaranteed to be better than the schedule in Figure 3(c), if the disjointness conditions $D_1 \cap D_3 = \phi$, $D_1 \cap D_4 = \phi$, $D_2 \cap D_3 = \phi$, $D_2 \cap D_4 = \phi$ are satisfied and $D_3 \cap D_4 \neq \phi$. In this case for the schedule of Figure 3(b) we have R_{12} disjoint from R_{34}, but in Figure 3(c) R_{13} is not disjoint from R_{24}.

More complex transformations are also possible. Further, transformations that do not satisfy the conditions of the theorem for only a subset of the variables can be used.

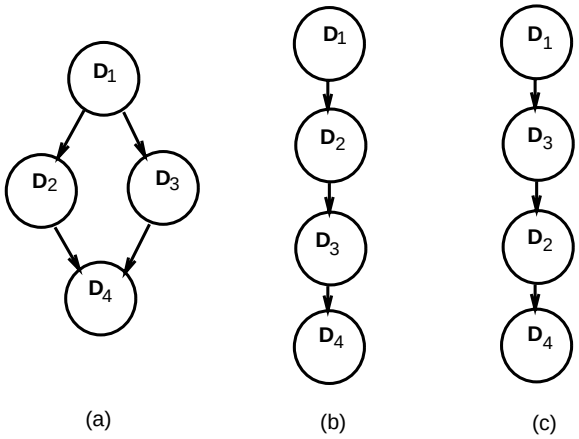


Figure 3: (a) A Dependency Graph, (b), (c) Schedules of Nodes

5 Algorithm for Program Transformations

Based on the theorem described in Section 3, we have developed an algorithm for program transformation to improve cache performance. The algorithm is based on reordering a program's memory reference stream such that the reordered memory reference stream satisfies the disjointness property. The algorithm can be applied at different levels of the program. It can reorder instructions within a basic block, transform loops, reorder blocks within a procedure, or reorder procedures within a program to improve hit rate for any cache that uses LRU replacement policy. The algorithm can be used by the compiler for reordering program code or for scheduling subroutine calls.

The algorithm schedules a dependence graph whose nodes can be basic blocks, procedures, or loops such that the conditions of the disjoint sequence theorem are maximally satisfied. It partitions the graph into two sets, where the accesses of one set are disjoint from the other, if possible, and recursively partitions and reorders the resulting sets. The recursive partitioning and reordering algorithm is given below:

Input:

1. A program to be transformed
2. A hierarchical dependency graph G with nodes representing the references, basic blocks, loops, or procedures. The edges represent the dependency or precedence among the nodes.

Output: A transformed program to improve cache performance
Algorithm: $Schedule(S)$

1. If the nodes in S can be partitioned into two sets S_1 and S_2 such that the accesses R_1 of S_1 and R_2 of S_2 satisfy the disjointness condition $R_1 \cap R_2 = \phi$.
   (a) $Schedule(S_1)$
   (b) $Schedule(S_2)$

2. Otherwise, partition S into two sets S_1 and S_2 based on the *Disjointness Metric*.
   (a) $Schedule(S_1)$
   (b) $Schedule(S_2)$

Disjointness Metric: Our approach is to keep the reordering transformations independent of the cache parameters and satisfy the disjointness property according to the theorems in the Section 3. But, with the dependency constraints, it is not always possible to reorder the reference streams into completely disjoint reference streams according to the theorems. In such cases, we use a disjointness metric and minimize it using the reordering transformations. The disjointness metric is a heuristic that approximates the complete disjointness condition of the theorems.

Our disjointness metric captures the disjointness of the two partitions of memory references S_1 and S_2. We minimize the disjointness metric: $abs(\|S_1\| - \|S_2\|) + (\|S_1 \cap S_2\|)$, where $\|S_i\|$ is the number of memory references in S_i. The first part of the disjointness metric $abs(\|S_1\| - \|S_2\|)$ balances the partition sizes $\|S_1\|$ and $\|S_2\|$ and the second part $(\|S_1 \cap S_2\|)$ reduces the overlap of S_1 and S_2.

6 Experimental Results and Analysis

We used some of the Spec95 benchmarks for our experiments. We analyzed the benchmarks to determine the dependence graph for the functions in the programs. In the benchmarks we considered, there was no opportunity to reorder the functions calls. We profiled the benchmarks to determine the functions that contributed the most in terms of the number of misses. We picked some of these functions to determine the dependence graph and reorder the blocks or loops. The hit rates for the functions that were transformed for the four Spec95 benchmarks `swim`, `tomcatv`, `wave5`, and `applu` are shown in Figure 4. In the first column we give the number of accesses of the transformed functions as a percentage of the total number of accesses. The column "Cache Size" indicates the cache size in bytes. We considered two cache sizes: 8K bytes and 16K bytes. The column "Assoc, Block" shows the associativity and block size in bytes. We considered the associativity 2 and 4 and block size 16 and 32. The column "Original Hit Rate (%)" shows the hit rate of the original code. The column "Transformed Hit Rate (%)" shows the hit rate of the transformed code. The column "Relative Improvement (%)" shows the percentage improvement in hit rate over the original code. The overall hit rate results for the four Spec95 benchmarks `swim`, `tomcatv`, `wave5`, and `applu` are shown in Figure 5. The results for these Spec95 benchmarks for different cache sizes and organization show that the transformed code performs better than the original code and in some cases there is significant improvement in hit rate.

7 Conclusions

In this paper we proved some theorems and showed the application of these theorems using some reordering transformations. We also showed a disjointness metric that is used by our algorithm to recursively perform the reordering transformation in presence of the dependency

Benchmark Function(s)	Cache Size	Assoc, Block	Original Hit Rate (%)	Transformed Hit Rate (%)	Relative Improvement (%)
swim	8K	2, 16	79.2383	95.2317	20.1839
	8K	2, 32	75.9248	91.7274	20.8134
transformed	8K	4, 16	92.3153	95.8974	3.8802
functions	8K	4, 32	82.2497	97.1300	18.0916
have 27.6%	16K	2, 16	94.4201	96.1583	1.8409
of total	16K	2, 32	83.5730	96.8736	15.9149
accesses	16K	4, 16	93.2062	96.1089	3.1142
	16K	4, 32	83.2812	97.2020	16.7154
tomcatv	8K	2, 16	90.2541	92.1447	2.0947
	8K	2, 32	90.8541	92.1089	1.3811
transformed	8K	4, 16	93.9420	94.0693	0.1355
functions	8K	4, 32	93.5937	94.8125	1.3022
have 93.4%	16K	2, 16	93.8268	94.6455	0.8725
of total	16K	2, 32	93.6977	94.6172	0.9813
accesses	16K	4, 16	93.9752	94.0788	0.1102
	16K	4, 32	93.6229	94.8285	1.2877
wave5	8K	2, 16	85.7735	86.7355	1.1215
	8K	2, 32	86.3241	87.5198	1.3851
transformed	8K	4, 16	88.7721	89.4833	0.8011
functions	8K	4, 32	89.6087	91.1452	1.7146
have 21.7%	16K	2, 16	89.8719	90.6470	0.8624
of total	16K	2, 32	90.7452	92.0300	1.4158
accesses	16K	4, 16	89.1097	90.1596	1.1782
	16K	4, 32	89.8363	91.5442	1.9011
applu	8K	2, 16	93.8072	93.8074	0.0004
	8K	2, 32	96.8070	96.8081	0.0011
transformed	8K	4, 16	93.9381	93.9386	0.0005
functions	8K	4, 32	96.9206	96.9224	0.0018
have 58.7%	16K	2, 16	98.5511	98.5821	0.0314
of total	16K	2, 32	99.1859	99.2197	0.0340
accesses	16K	4, 16	98.4721	98.5031	0.0314
	16K	4, 32	99.1460	99.1803	0.0345

Figure 4: Hit rates of the transformed function(s) of the programs

Benchmark	Cache Size	Assoc, Block	Original Hit Rate (%)	Transformed Hit Rate (%)	Relative Improvement (%)
swim	8K	2, 16	86.1321	90.7588	5.3716
	8K	2, 32	85.0540	89.4250	5.1390
	8K	4, 16	89.7846	90.9963	1.3495
	8K	4, 32	85.9929	90.2829	4.9887
	16K	2, 16	95.2765	95.7724	0.5204
	16K	2, 32	91.9342	95.5598	3.9436
	16K	4, 16	95.1871	96.0330	0.8886
	16K	4, 32	91.1608	95.0330	4.2476
tomcatv	8K	2, 16	90.8852	92.6592	1.9519
	8K	2, 32	91.4221	92.6124	1.3019
	8K	4, 16	94.3362	94.4619	0.1332
	8K	4, 32	94.0186	95.1569	1.2107
	16K	2, 16	94.2299	95.0002	0.8174
	16K	2, 32	94.1113	94.9747	0.9174
	16K	4, 16	94.3698	94.4733	0.1096
	16K	4, 32	94.0425	95.1741	1.2032
wave5	8K	2, 16	95.4915	95.5878	0.1008
	8K	2, 32	96.1502	96.2983	0.1540
	8K	4, 16	96.3550	96.4212	0.0687
	8K	4, 32	97.0162	97.2719	0.2635
	16K	2, 16	96.8037	96.8926	0.0918
	16K	2, 32	97.4031	97.6111	0.2135
	16K	4, 16	96.7395	96.8821	0.1474
	16K	4, 32	97.2599	97.5543	0.3026
applu	8K	2, 16	95.7379	95.7380	0.0001
	8K	2, 32	97.8144	97.8224	0.0081
	8K	4, 16	95.8144	95.8146	0.0002
	8K	4, 32	97.8831	97.8899	0.0069
	16K	2, 16	98.5416	98.5587	0.0173
	16K	2, 32	99.2105	99.2302	0.0198
	16K	4, 16	98.4975	98.5157	0.0185
	16K	4, 32	99.1932	99.2115	0.0184

Figure 5: Overall hit rates for the original and transformed programs

constraints. Our preliminary results show significant improvement in some cases. We plan to incorporate this reordering transformation approach into a compiler framework and fully automate this approach to get results for a bigger set of benchmarks and programs.

References

[1] Jacqueline Chame and Sungdo Moon. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 492–499, Rhodes, Greece, June 20-25 1999.

[2] Matteo Frigo, Charles E. Leiserson, Harold Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *the 40th Annual Symposium on Foundations of Computer Science, FOCS '99*, New York, NY, USA, 17-18 October 1999.

[3] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[4] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Systems*, pages 228–239, San Jose, CA, USA, October 2-7 1998.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[6] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality using Loop and Data Transformations in an Integrated Framework. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, MICRO-31*, pages 285–296, November 30 - December 2 1998.

[7] M. S. Lam, E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, USA, April 8-11 1991.

[8] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[9] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 425–433, Rhodes, Greece, June 20-25 1999.

[10] H. Prokop. *Cache-Oblivious Algorithms.* Master's Thesis, Massachusetts Institute of Technology, June 1999.

[11] Yonghong Song and Zhiyuan Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, GA, USA, May 1-4 1999.

[12] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 26-28 1991.

[13] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO-29*, pages 274–286, December 2-4 1996.