The Complexity of Scheduling for Data Cache Optimization

Devidas Gupta, Brian Malloy and Alice McRae
Department of Computer Science
Clemson University
Clemson, SC, 29634-1906
gupta@cs.clemson.edu

Abstract

The performance of the memory hierarchy, and in particular the data cache, can significantly impact program execution speed. Thus, instruction reordering to minimize data cache misses is an important consideration for optimizing compilers. In this paper, we prove that the problem of instruction reordering for data cache miss minimization belongs to the class of NP-complete problems. The framework that we develop for the proof exposes the symbiotic relationship among the references to the cache. This symbiosis exists because a single cache reference lengthens the life span of its neighbors in the cache and thus provides opportunity for additional cache hits through reference to the neighbors. We present a greedy heuristic designed to exploit this symbiotic relationship to improve data cache performance for general purpose programs. Experiments with a prototype implementation of the heuristic show that we can improve data cache performance in many cases.

1 Introduction

Cache memories have become an important intermediary between relatively slow main memory and the fast processor of todays computer systems. To illustrate the importance of data cache management, consider that a register miss together with a cache hit is likely to result in a delay of a single cycle. However, a register miss together with a cache miss is likely to result in a delay of ten or twenty cycles, and perhaps as many as several hundred cycles. Thus, instruction reordering to minimize the number of data cache misses has become an important consideration for optimizing compilers.

Recent research in cache optimization has focused on improving the *instruction cache* reference pattern [5, 8, 9, 10, 11, 13, 14]. The goal for instruction cache optimization is to reorder a sequence of instructions so that the instructions referenced the most are likely to be found in the cache. Some research has addressed the problem of improving the *data cache* reference pattern [3, 6, 12, 16, 17]. The goal for data cache optimization is to generate a sequence of instructions where the operands referenced the most are likely to be found in the cache. Wolfe and Lam have improved data cache performance by transforming loops using techniques such as *interchange*, *skewing* and *reversal* [16, 17]. They present an algorithm that combines the above loop transformations to improve the data locality for numerical algorithms that use matrices. Ju and Dietz present techniques to modify the data layout of a program to improve the data cache accesses in loops[6]. They use an interference graph to provide information about interactions between data layouts and the loops

that reference the data. The interference graph is pruned to reduce complexity and a search of the graph produces the new data layout.

Previous research for improving data cache performance has focused on optimizing the cache references in loops, especially predictable loops such as those found in scientific applications. The problem of optimizing data cache references for general purpose programs, at a fine-grained level, has not been adequately addressed. However, general purpose programs have potential for excellent speedup through improved data cache performance. The ideal for data cache management is to reorder instructions so that the number of cache misses will be minimal. However, in this paper, we prove that the problem of instruction reordering for data cache miss minimization belongs to the class of NP-complete problems. The framework that we develop for the proof exposes the symbiotic relationship among the references to the cache. This symbiosis exists because a single cache reference lengthens the life span of it's neighbors in the cache and thus provides opportunity for additional cache hits through reference to the neighbors. We present a greedy heuristic that exploits this symbiotic relationship to improve data cache performance.

In the next section we detail our representation of the problem and introduce *schedule graphs*. Section 3 defines the *Cache Optimization* problem and provides a proof of NP-completeness. We present a heuristic for improving the data cache reference pattern in section 4 and conclude in section 5.

2 Problem Description

In this section, we develop a program representation called the *schedule graph* that we use in our proof and in our heuristic. We then motivate the problem under study with an example.

2.1 Problem Representation

A *schedule graph* is the complete representation that we use for a basic block[1]. A *schedule graph* is a directed graph $G(V, E)$ consisting of a set of vertices V and a set of edges E. Each vertex $v \in V$ represents an instruction. Each edge $(v, w) \in E$ represents a dependence between vertices v and w. There are two kinds of dependences represented by E and these dependences partition E into two sets E_p and E_s; we will now discuss these two sets.

A data dependence edge from instruction v to instruction w is an edge from a definition in v that is used by w; we call this set of dependence edges E_p and the graph $G(V, E_p)$ is a directed acyclic graph or dag[1]. A constraint edge represents a constraint imposed by the schedule[2] that G represents. That is, there is an edge (v, w) from instruction v to instruction w if v precedes w in the given schedule S; we call this set of

[1] A basic block is a sequence of statements such that the only entrance to the block is through the first statement and the only exit from the block is through the last statement[1].

[2] A schedule is an ordered set of nodes in the *schedule graph* G and is obtained by traversing G, following the scheduling edges.

scheduling edges E_s. Thus, the set of edges in G are $E = E_p \cup E_s$. The heuristic that we present in section 4 accepts $G(V, E_p)$ as input and adds scheduling edges to produce the graph $G(V, E_p \cup E_s)$.

2.2 Characteristics of schedule graphs - the graph invariants

A *schedule graph*, G, has the following characteristics:

- G has three types of vertices.
  - start vertex : vertex v is a *start vertex* if $pred(v) = \emptyset$. G has exactly one start vertex.
  - finish vertex : vertex v is a *finish vertex* if $succ(v) = \emptyset$. G has exactly one finish vertex.
  - interior vertices : vertex v is an *interior vertex* if v is neither a *start* nor a *finish* vertex.
- G is acyclic. This property follows from the observation that an instruction cannot be scheduled until all of its operands have been computed.
- G is connected. This property follows from the observation that all instructions must be scheduled regardless of their precedence constraints.
- G may be a *multi* digraph. There may be more than one directed edge from one node to another. To illustrate, the schedule graph in Figure 1 contains node g and node h; there are two edges from node g to node h.

2.3 Computation of cost for a schedule

In order to evaluate our computed schedule, we incorporate the cost of a cache miss into the *schedule graph*. We associate with every vertex v_i, in a *schedule graph* G, a cost C_i representing the cost of that vertex. Included in C_i, is the sum of the costs of the scheduling the predecessors of v_i so that C_i represents the cost of executing S up to vertex i. Furthermore, the cost associated with the *finish vertex* is the cost of the schedule. The cost C_i associated with vertex v_i is the sum of (1) the cost to access the operands of v_i,denoted by t_i, (2) the cost to execute v_i (assuming the operands of v_i have been fetched), denoted by e_i, and (3) the sum of the costs of scheduling predecessors of v_i.

Thus, the total cost[3] of a vertex v_i in G is:

$$C_i = t_i + e_i + \sum C_j \mid j \in pred_s(i) \quad (1)$$

[3]Note that the cost of a vertex depends on the scheduling predecessors (and not the precedence predecessors).

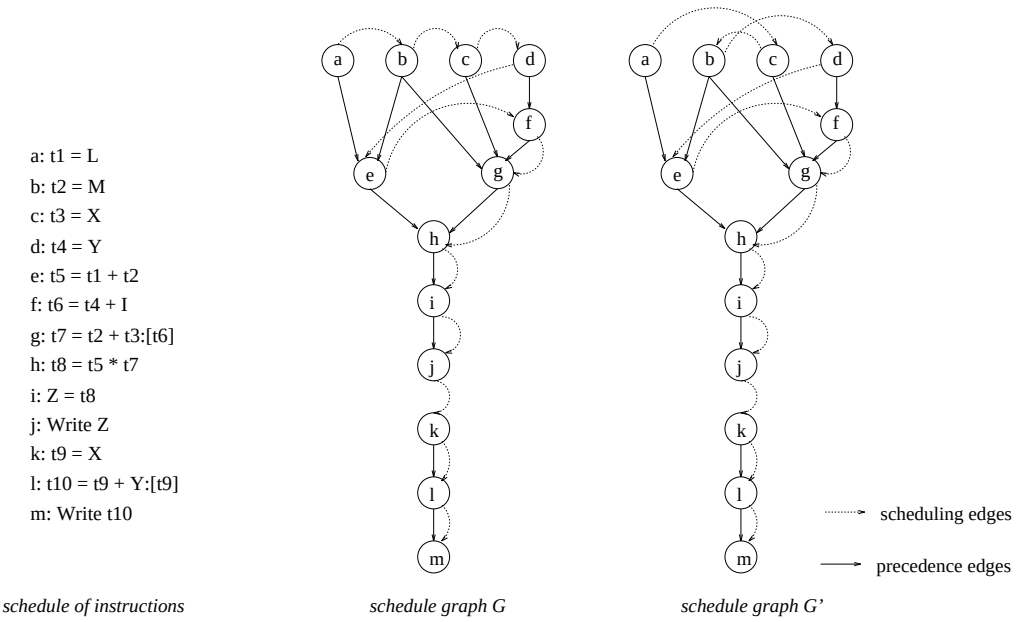

Figure 1: **The schedule graph**. This figure illustrates a set of instructions or "a schedule of instructions", a *schedule graph* G for the instructions, and a second *schedule graph* G' with the first four instructions reordered

2.4 An Example

We now illustrate the representation presented in the previous section with a motivating example. Figure 1 illustrates a sequence of instructions, a *schedule graph* for the sequence of instructions, and a second *schedule graph* with the first four instructions reordered. In the graph, a is a *start* vertex and m is a *finish* vertex. Our goal is to reorder the instructions so that finish vertex m has a minimum scheduling cost. For brevity, we compute the costs associated with the first four vertices in the schedule. The computed costs (we assume the execution cost of each instruction is a constant, EC) associated with the four vertices a, b, c and d are:

$$
\begin{aligned}
C_a &= t_a + EC, \\
C_b &= t_b + EC + C_a, \\
C_c &= t_c + EC + C_b, \\
C_d &= t_a + EC + C_c
\end{aligned}
$$

To show how reordering the instructions can reduce the number of cache misses for a schedule, assume that:

- The memory-cache relationship shown in Figure 2 holds for the schedule given in Figure 1. That is, the line in memory containing L and X, and the line in memory containing M and Y, both map to the same line in the cache. This implies that at any given time, either the line containing L and X or the line containing M and Y, but not both the memory lines, can be present in the cache.

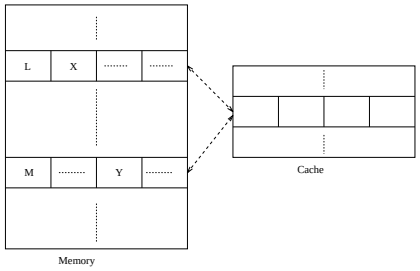

Figure 2: **The memory - cache relationship.** This figure illustrates the memory-cache relationship for the operands of the instructions in the schedule graph shown in Figure 1. The line in memory containing the variables L and X, (the operands for instructions a and c), and the line in memory containing the variables M and Y, (the operands for instructions b and d), both map to the same line in the cache. Thus, either the operands for instructions a and c or the operands for instructions b and d, but not both, can exist in the cache at any given time.

- The cost of fetching the operands of an instruction on a cache hit is C_{hit} and the cost of fetching the operands of an instruction on a cache miss is C_{miss}. Assume also that both C_{hit} and C_{miss} are constants and that $C_{hit} < C_{miss}$.

- There is no preloading, so that the cache does not contain any operands of any instructions in the schedule before the schedule begins.

Under the above assumptions and for the schedule presented in Figure 1, a causes a miss, b causes a miss, c causes a miss, and d causes a miss. The costs associated with vertex d, which represents the cost of the schedule $a + b + c + d$, is

$$\begin{aligned} C_d &= C_{miss} + EC + (C_{miss} + EC + (C_{miss} + EC + (C_{miss} + EC))) \\ &= 4 \cdot C_{miss} + 4 \cdot EC \end{aligned}$$

Assume that we transform *schedule graph* G into *schedule graph* G' as shown in Figure 1. In the new schedule, the following relationships hold:

$$(a, c), (c, b), (b, d), (d, e) \in Es$$

In the transformed schedule G', a causes a miss, c causes a hit, b causes a miss, and d causes a hit. The cost of the vertex d' is given by:

$$\begin{aligned} C'_d &= C_{hit} + EC + (C_{miss} + EC + (C_{hit} + EC + (C_{miss} + EC))) \\ &= 4 \cdot EC + 2 \cdot C_{miss} + 2 \cdot C_{hit} \end{aligned}$$

Comparing C_d and C'_d, we see that the transformed schedule S' is better than S since $C_{hit} < C_{miss}$.

3 Complexity of the Scheduling

We now formulate the decision problem to reflect the number of cache misses in the schedule imposed in a *schedule graph*. We investigate the complexity of the problem and show that the decision problem is in the class of *NP-complete* problems.

3.1 Problem definition

The decision problem for *C*ache *Opt*imization, $COPT$, is presented as follows.

3.1.1 The $COPT$ problem

COPT

Instance :

A *schedule graph*, G(S, $E_s \cup E_p$),
a cache with n lines,
a memory with l lines, $l = n^2 + n$
a set ϕ of triplets (α, β, γ) representing the mapping of the operands of each instruction to the memory and the cache. The operands of α reside in line β of memory and are fetched from line γ in the cache.
$\alpha \in S$, $1 \leq \beta \leq l$, $1 \leq \gamma \leq n$,
a non-negative integer k.

Question :

Does there exist a schedule in the *schedule graph* $G'(S', E'_p \cup E'_s)$ such that $G'(S', E'_p)$ is isomorphic to $G(S, E_p)$ and the number of cache misses in the schedule imposed by $G' \leq k$?

3.2 $COPT$ is in NPc

We now prove that $COPT$ is NP-complete. The proof follows from a reduction of *Exact Cover by 3 sets*, *X3C*, to $COPT$.

3.2.1 Polynomial verifiability of $COPT$

Lemma 1 $COPT \in NP$.

Proof: Clearly $COPT \in NP$. A nondeterministic algorithm need only guess at the schedule and then verify in polynomial time whether the schedule imposed leads to k or less number of cache misses.

3.2.2 The $X3C$ problem

X3C

Instance :

A finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X.

Question :

Does C contain an *exact cover* for X, that is, a sub-collection $C' \in C$ such that every element of X occurs in exactly one member of C' ?

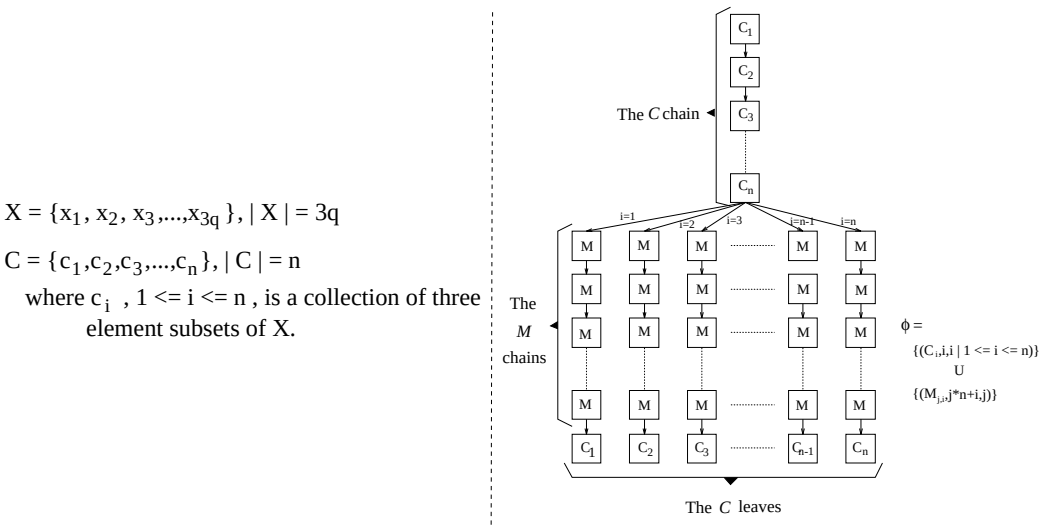

Figure 3: **Construction of the schedule graph from a given instance of X3C.** This figure depicts an arbitrary instance of X3C and the transformed instance of COPT. Constructing the COPT instance from the X3C instance is a 4 step procedure and involves building the C chain (at the top of the *schedule graph*), building the M chains, adding the C leaves and introducing ϕ.

3.2.3 A construction to transform $X3C$ to $COPT$

Lemma 2 *An arbitrary instance of $X3C$ can be transformed to an instance of $COPT$.*

Proof: We prove that an arbitrary instance of $X3C$ can be transformed to an instance of $COPT$ by presenting a construction that builds an instance of $COPT$ from a given instance of $X3C$.

Let an arbitrary instance of $X3C$ be given by

$X = \{x_1, x_2, x_3, \ldots, x_{3q}\}, |X| = 3q$

$C = \{c_1, c_2, c_3, \ldots, c_n\}, |C| = n$, where c_i, $1 \leq i \leq n$ is a collection of three element subsets of X.

We now construct an instance of $COPT$ using our arbitrary instance of $X3C$. The construction is performed in four steps and results in a tree structure for the *schedule graph* in $COPT$. Figure 3 illustrates the construction used to transform an arbitrary instance of $X3C$ to an instance of $COPT$.

Construction 1 *The C chain.*

For every clause c_i in the collection C, we construct a vertex C_i in G such that C_1 is the *root* vertex in G and $(C_i, C_{i+1}) \in E_p$, $1 < i < n$.

Construction 2 *The M chains.*

For every clause c_i in the collection C, we construct an M chain. An M chain is a chain of vertices that results from the comparison of a particular c_i to the other subset in C, but not itself. G has a vertex $M_{j,i}$ such that $\{M_{j,i} | c_i \cap c_j \neq \emptyset$ and $i \neq j$; $c_i, c_j \in C\}$ i.e. for every pair of subset in the collection C that have a non-empty intersection, we add a node M to G. Within an M chain, the vertices are connected together by precedence edges. These edges are added as follows. Let l be the total number of vertices in an M chain. Then, $(M_{x,i}, M_{x+1,i}) \in E_p$, $1 \leq x < l$. These M chains are then linked to the vertex C_n constructed in the previous step (Construction 1) by adding a precedence edge from C_n to the vertex at the top of every M

chain. We thus have n number of M chains, each linked to the vertex C_n and therefore there are n number of edges emanating from vertex C_n. In figure 3, the nodes denoted by Ms have been constructed in this phase.

Construction 3 *The C leaves.*

Lastly, we add another n nodes, $C_1 \ldots C_n$ corresponding to the n subsets in C such that every M chain terminates in a leaf labeled C. The precedence edges are added as follows. $(M_{x,i}, C_i) \in E_p$ where $M_{x,i}$ is the last M vertex in the ith M chain. Let m be the total number of such M vertices.

Construction 4 *The n line cache, l line memory and ϕ.*

In the last step we add a cache with n lines, a memory with $l = n^2 + n$ lines and introduce ϕ.

The set ϕ is constructed as follows :

$$\phi = \{(C_i, i, i | 1 \leq i \leq n)\} \cup \{(M_{j,i}, j \cdot n + i, j)\}$$

We thus restrict the memory locations for the operands of the vertices such that no two C vertices or no two M vertices are allocated within the same memory line. No M vertex shares a memory line in common with a C vertex. Thus the operands of all the vertices labeled C and the vertices labeled M are assigned distinct memory locations.

We thus obtain an instance of $COPT$ by transformation of an arbitrary instance of $X3C$.

3.2.4 Complexity of the transformation

We now show that the construction presented in section 3.2.3 is polynomial.

Lemma 3 *The construction presented is polynomial.*

Proof:

The construction of the C chain involves the construction of n vertices and this can be done in linear time. Building the M chains and linking them to the C_n vertex requires the construction of m nodes and m can be at most $2 \cdot \begin{pmatrix} n \\ \\ 2 \end{pmatrix}$. The construction of the edges in each M chain requires a set intersection operation for all elements in C and this can be done in time polynomial in n. The third construction step once again adds n nodes and hence this step can be done in linear time. And the last step takes at most $2 * n + m$ steps, since we construct a mapping for every vertex in the *schedule graph*.

Thus, the construction is polynomial in n, where $n = |C|$, the size of the collection C.

3.2.5 Equivalence of a solution for $X3C$ and that for $COPT$

From the construction presented in section 3.2.3 we know that we can transform an arbitrary instance of $X3C$ to an instance of $COPT$ in polynomial time. We now show that such a transformation guarantees that the $COPT$ instance is a *YES* instance if and only if the corresponding $X3C$ instance is a *YES* instance (and similarly for a *NO* instance).

Lemma 4 *The number of cache misses, k, in the constructed instance of $COPT$ is given by $k \leq 2 \cdot n + m - q$ if the given instance of $X3C$ has an exact cover (of size q).*

Proof:

For the proof, we present a technique for obtaining the scheduling edges in the constructed instance of

$COPT$.

We begin at the root of the tree, i.e., at the vertex C_1, which has no precedence predecessors, and add scheduling edges to the C chain as follows: $(C_i, C_{i+1}) \in E_s$, $1 \leq i < n$. We then traverse the M chains in a prioritized fashion : we traverse all those chains first that have a C leaf corresponding to a clause in the exact cover for X. Within a chain, we traverse all the vertices beginning at the top of the chain (adjacent to the interior C_n) and proceed to the bottom of the chain ending at the C leaf. Finally, we traverse all the M chains each of which has a C leaf that does not form the sub-collection that covers X.

We have m vertices labeled M. These m vertices cause m number of misses since their construction and the mapping function ensures that no two such vertices map to the same cache line, from the same memory line, and that no M vertex has a memory line in common with any of the C vertices, which are n in number.

The traversal of the C chain results in n number of cache misses. The traversal of the M chains results in m number of cache misses. These C nodes are once again encountered at the bottom of every M chain and hence they may cause another n misses. A C_i vertex that occurs at the bottom of a $M_{j,i}$ chain will have a miss, if and only if, there existed in the schedule a M node that mapped to line i and was scheduled before the C_i leaf vertex. The M node thus overwrote the ith line in the cache. But, in our traversal, we always visit the chains with the C leaf vertices corresponding to the subsets in the exact cover first and we do have an exact cover. Our construction and mapping guarantees that in such a situation, we never add a M vertex such that it conflicts with a cache line due to a C leaf vertex corresponding to the exact cover. (Recall that a vertex $M_{j,i}$ is added only if a clause c_i has a member in common with c_j and by definition of exact cover, the q subsets forming the exact cover, do not have any elements in common). Our construction (and mappings) save us q cache misses if there exists an exact cover.

Thus, the total number of misses obtained is

$\leq n + m + (n - q)$

$\leq 2 \cdot n + m - q$

Thus, the claim holds.

Lemma 5 *An instance of $X3C$ has an exact cover if the number of cache misses, k, in the constructed instance of $COPT$ is given by $k = 2 \cdot n + m - q$.*

Proof:

The instance of $X3C$ will have an exact cover if the instance of $COPT$ has the number of cache misses given by , $k = 2 \cdot n + m - q$. For finding the exact cover, we traverse the schedule imposed by the *schedule graph* in the constructed $COPT$ instance and pick up those C leaf vertices that do not cause cache misses. Now, there are exactly q such C leaf vertices, and the subsets in the $X3C$ instance that correspond to these leaf vertices form the exact cover. Since these C leaf vertices never caused a miss, it must be that the M nodes encountered in the schedule, prior to the scheduling of these C leaf vertices, never had a memory reference that mapped their operands onto the same lines as that of a C leaf vertex. Such a restraint could have been imposed only if the corresponding subsets in the $X3C$ instance did not share any element from X (recall the construction of the M vertices). The subsets corresponding to these C leaf vertices thus form an exact cover for the set X.
Hence the claim holds.

3.2.6 NP-completeness of $COPT$

Lemma 6 *$COPT$ is in NPc.*

Proof: Proof is immediate from lemmas 1, 2, 3, 4 and 5 and the definition of NP-completeness[4].

3.3 An Example of transforming an instance of $X3C$ to an instance of $COPT$

Figure 4 presents an example which illustrates the construction of an instance of $COPT$ from a given instance of $X3C$. Let us suppose that we were given the following instance of $X3C$.

$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$C = \{(x_1, x_2, x_4), (x_1, x_3, x_4), (x_2, x_5, x_6), (x_2, x_3, x_5)\}$

$|X| = 6$, $3q = 6$, $q = 2$

$|C| = n = 4$

We have 4 subsets in our instance of $X3C$, $c_1 \ldots c_4$ where

$c_1 = (x_1, x_2, x_4)$

$c_2 = (x_1, x_3, x_4)$

$c_3 = (x_2, x_5, x_6)$

$c_4 = (x_2, x_3, x_5)$

We now construct the $COPT$ instance by using the four-step approach presented in section 3.2.3.

Construction - Step 1

For each of the 4 subsets in C, we construct a vertex and obtain the chain of vertices $c_1 \ldots c_4$ and the corresponding edges as shown in figure 4.

Construction - Step 2

Now, we construct the M nodes. We add a node $M_{j,i}$ for every pair of subsets, c_i, c_j, if their intersection is non-empty and $i \neq j$. We thus have,

$c_1 \cap c_2 = x_1 \neq \emptyset \Rightarrow$ a vertex M_{21}

$c_1 \cap c_3 = x_2 \neq \emptyset \Rightarrow$ a vertex M_{31}

$c_1 \cap c_4 = x_2 \neq \emptyset \Rightarrow$ a vertex M_{41}

$c_2 \cap c_1 = x_1 \neq \emptyset \Rightarrow$ a vertex M_{12}

$c_2 \cap c_4 = x_3 \neq \emptyset \Rightarrow$ a vertex M_{42}

$c_3 \cap c_1 = x_2 \neq \emptyset \Rightarrow$ a vertex M_{13}

$c_3 \cap c_4 = x_2 \neq \emptyset \Rightarrow$ a vertex M_{43}

$c_4 \cap c_1 = x_2 \neq \emptyset \Rightarrow$ a vertex M_{14}

$c_4 \cap c_2 = x_3 \neq \emptyset \Rightarrow$ a vertex M_{24}

$c_4 \cap c_3 = x_2 \neq \emptyset \Rightarrow$ a vertex M_{34}

These vertices are the chain of branches emanating from the vertex c_n as shown in figure 4.

Construction - Step 3

Now in the third step we add another n vertices, $c_1 \ldots c_4$ at the bottom of the M vertices as shown in figure 4.

Construction - Step 4

All that remains is the introduction of the cache, memory and ϕ. We introduce a cache with 4 lines and a memory with 20 lines. ϕ is as follows.

$$\begin{aligned}\phi = \{&(C_1, 1, 1), (C_2, 2, 2), (C_3, 3, 3), (C_4, 4, 4)\} \cup \\ &\{(M_{2,1}, 9, 2), (M_{3,1}, 13, 3), (M_{4,1}, 13, 4), \\ &\quad (M_{1,2}, 6, 1), (M_{4,2}, 18, 4), \\ &\quad (M_{1,3}, 7, 1), (M_{4,3}, 19, 4), \\ &\quad (M_{1,4}, 8, 1), (M_{2,4}, 12, 2), (M_{3,4}, 16, 3)\}\end{aligned}$$

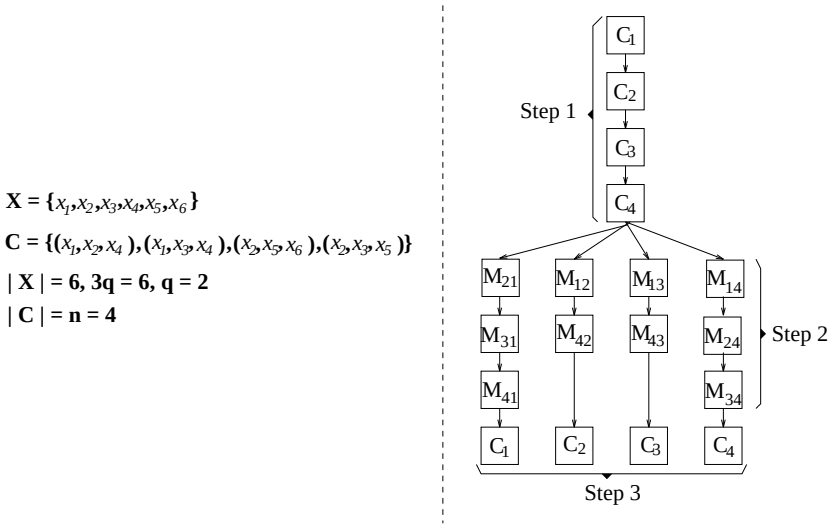

Figure 4: **Example Construction** This figure shows a specific instance of the $X3C$ problem and the equivalent instance of the $COPT$ problem. The C chain at the top of the figure is constructed in the first step. The second step of the construction results in the M chains. The C leaves are added in the third step of the construction.

Thus, we have taken a given instance of $X3C$ and constructed an instance of $COPT$ using the construction steps introduced previously.

4 A Heuristic for Improving the Data Cache Reference Pattern

The ideal approach to solving the the problem of optimizing the data cache reference pattern is to design an algorithm that uses a dag as input and produces a schedule as output, where this schedule minimizes the number of cache misses. However, since the scheduling problem that we address is NP-complete, we now present the the design of a heuristic for producing good schedules.

4.1 The heuristic

The heuristic is a greedy approach where we try to schedule consecutively, grouping together wherever possible, all those statements containing operands that map to both the same line in memory and in the cache. The heuristic illustrated in Figure 5, accepts a dag as input and produces a *schedule graph* as output. The nodes of the dag represent assembly language instructions where each instruction contains at most one memory reference. We label each node in the dag with an ordered pair, where the first element of the pair represents the memory line for the operand and the second element of the pair represents the cache line where the operand will be mapped. The edges in the dag represent a partial ordering of the nodes imposed by the data dependencies among the instructions. A sample dag for Figure 3 is shown in Figure 7 with ordered pairs attached to each node.

algorithm BuildScheduleGraph
input $G(S, E_P)$: a dag containing **n** nodes. Associated with every node is an ordered pair *(x,y)* representing the memory and cache line for the operand of the node. The operand of the node resides in memory line x and is fetched from cache line y.
output $G(S, E_P \cup E_S)$: a schedule graph
local R : a list
local C : cache entry table. This table stores ordered pairs associated with the nodes of G
local PreviousNode : ptr to a node in dag G_D

```
begin BuildScheduleGraph

    initialize the ready list R with nodes having no predecessors;

    initialize all entries in C to (0,0);

    for i := 1 to n loop

        if there is a node n_i in R whose ordered pair matches
            an entry in C,  then schedule n_i
        elsif there are two nodes n_i, n_j in R, whose ordered pairs match then
            schedule n_i, the node that occurs first in list R
        elsif there are two nodes n_i, n_j in R, whose ordered pairs conflict then
            schedule n_i, the node that occurs first in list R
        else schedule any node n_i in R, preferring (1) a node that doesn't conflict with a cache entry
            or, (2) a node at the highest level of the dag
        end if;

        if this is the first iteration of the loop then
            PreviousNode := n_i;
        else
            construct an edge from PreviousNode to n_i;
            PreviousNode := n_i;
        end if;

        update the cache entry table C, to the ordered pair associated with n_i;

        insert any nodes into R whose predecessors are scheduled;
    end loop;

end  BuildScheduleGraph;
```

Figure 5: **Algorithm** `BuildScheduleGraph`**.** This figure depicts the algorithm to build a *schedule graph* from a dag. The algorithm maintains a ready list of nodes and iteratively schedules the nodes, attempting to minimize the cache misses. A cache entry table helps determine conflicts.

The algorithm, `BuildScheduleGraph`, shown in Figure 5, is a version of *list scheduling* [2]. `BuildScheduleGraph` maintains a ready list, **R**, where a node is inserted into **R** if it is ready to execute, that is, all of its predecessors are scheduled. `BuildScheduleGraph` also uses a cache entry table **C**, for designating the current line of memory that has been inserted in the cache, and variable `PreviousNode` to assist in constructing edges in the *schedule graph*. `PreviousNode` points to the last node that was scheduled so that an edge can be constructed from the last node to the current node being scheduled.

In the **for** loop of `BuildScheduleGraph`, edges in the *schedule graph* are constructed, with an edge inserted into the graph for each iteration except the first. The **if-elsif** structure of `BuildScheduleGraph` chooses a node to be scheduled, from the ready list, based on the following criteria:

1. Choose a node from R whose ordered pair matches an entry in C, since this node will be a cache hit.

C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]
(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)

C: the Cache Entry Table

R = {a, b, c, d, k}

R: the Ready List

Figure 6: **Init-Values.** The initial values of **C**, the cache entry table, and **R**, the ready list for the example shown in figure 7 .

---

2. Choose a node from R whose ordered pair matches another node in R, since there will be a hit when scheduling the second node on the subsequent iteration of the **for** loop.
3. We have exhausted our supply of nodes that will cause a hit so we now attempt to separate misses by as many cycles as possible. Thus, our third choice schedules one of two nodes whose ordered pairs conflict[4].
4. Lastly, pick any node from the ready list preferring one that doesn't conflict with a current cache entry, or a node at the highest level of the dag.

Having chosen a node for scheduling, the remaining actions in the **for** loop of `BuildScheduleGraph` are to insert an edge into the *schedule graph* (except for the first node being scheduled), and update the cache entry table **C** and the ready list **R**. Loop iterations continue until all nodes in the dag have been scheduled. Since the **if-elsif** structure chooses exactly one node during every iteration, the **n** nodes of the input dag are guaranteed to be scheduled.

To illustrate the behavior of `BuildScheduleGraph`, we now model the actions of this heuristic for the dag in Figure 7. For illustration, we assume a data cache with seven lines and we depict the initial values of the cache entry table and the ready list in Figure 6. In the first loop iteration, no lines have been inserted into the cache, so there is no match for entries in R and C so that the **if** branch of the **if-elsif** structure is not taken. However, nodes a and c, and nodes b, d and k have matching ordered pairs associated with them; we assume that node a appears first in R so that nodes a and c are the first match found by the first **elsif** branch of the **if-elsif** structure. Thus, we assume that `BuildScheduleGraph` schedules node a, the current schedule is $S = \{a\}$, and that entry C[1] of the cache entry table is updated to $(1024, 1)$. For this first iteration of the loop, no edge is added to the schedule graph and no nodes are added to ready list R.

During the second iteration of the loop in `BuildScheduleGraph`, the ordered pair associated with node c in R, matches entry C[1]. Thus, node c is scheduled, $S = \{a, c\}$, and a scheduling edge is inserted from node

[4]A conflict occurs when two different memory lines are mapped to the same cache line

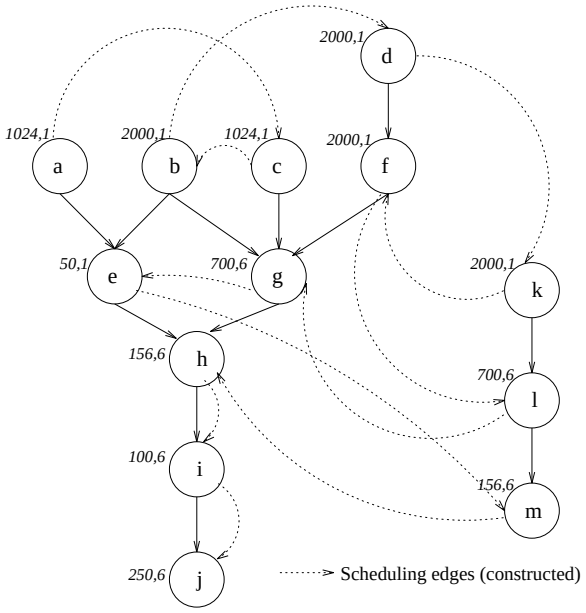

Figure 7: **Example dag.** Figure shows the sample dag used for modeling the heuristic **BuildScheduleGraph**. The dotted edges represent the scheduling edges and are added as the algorithm **BuildScheduleGraph** progresses. The final schedule is obtained by traversing the scheduling edges from node a to node j, thus obtaining the schedule, $S = \{a, c, b, d, k, f, l, g, e, m, h, i, j\}$ that results in 7 cache misses (as compared to 12 cache misses in the original schedule of figure 1).

---

a to node c. During the third iteration, the ordered pairs associated with nodes b, d, and k match; we assume that node b is scheduled since it appears before d and k in our list. Furthermore, an edge is constructed from c to b, $S = \{a, c, b\}c$, C[1] is updated to (2000, 1), and node e is inserted into ready list R since its predecessors are scheduled. After 13 iterations through the loop, the final schedule is $S = \{a, c, b, d, k, f, l, g, e, m, h, i, j\}$. The ordering of instructions listed in Figure 1 results in 12 data cache misses if the memory and cache are mapped as illustrated in Figure 7; however, using `BuildScheduleGraph` to obtain the ordering above, there are only 7 misses.

The running time of `BuildScheduleGraph` is dominated by the **for** loop and the time to search the ready list, **R**. In the worst case, ready list **R** may contain all of the nodes in the dag, since the program may not contain any data dependences. Assuming **n** statements in the program, the time to search ready list **R** is O(**n**) and, since the list might be traversed for each statement (in the **for** loop), `BuildScheduleGraph` may run in quadratic time.

However, since virtually all meaningful programs contain data dependences, ready list **R** is unlikely to contain all of the program statements. `BuildScheduleGraph` is intended to be applied to the straight line code found in the basic blocks of a program. Although research has shown that basic blocks contain few

statements, usually four to seven statements[15], they may contain as many as 200 if loops are unrolled[7]. We have found that, in practice, `BuildScheduleGraph` has little appreciable affect on the running time of the optimizer.

4.2 Experimentation & Results

In order to evaluate the performance of the heuristic, we designed and implemented a prototype. Preliminary results from experiments with the prototype are encouraging. The test suite consists of six programs with memory references disambiguated. By running the programs through a cache simulator before and after reordering, we obtained an estimate of the program improvement. Table 1 shows the test suite used in the experiments along with the number of lines in each program (the number of nodes in the *schedule graph*) and the number of variables.

Table 2 shows the average length of the ready list for each program. This gives a measure of the effort expended in searching for a node to schedule as well as the opportunity for reordering provided to the scheduler. For example, the average number of nodes in the ready list for `fib` was `1.05`, indicating that most of the time the heuristic did not have a choice and was forced to pick the only available node in the ready list. The length of the ready list depends on the precedence constraints in the *schedule graph*, and a larger number of constraints result in a lower number of ready nodes.

Table 3 presents the results of our preliminary experiments. The cache size varied from `8` to `128` bytes. The column labeled `Org.` shows the number of cache misses in the original code (for a given cache size) while the column labeled `Reord.` shows the number of cache misses in the reordered code (for the same cache size).

Our experiments indicate good performance gains for large programs or for programs that provide opportunity for reordering. Table 2 gives an indication of the opportunity that the heuristics had for optimizing since it lists the average number of nodes in the ready list. When the ready list contained 9.68 nodes (or more) the heuristic was able to substantially reduce the number of misses. For small programs or programs

Program Name	**Description**	**Number of lines**	**Number of variables**
`dual`	A dual interlocked dag	107	32
`fft`	A complete binary tree dag	190	127
`fib`	Fibonacci numbers	20	10
`mm`	Matrix multiplication - 3 x 3	120	27
`pyramid`	A grid dag	30	21
`whet`	The Whetstone benchmark program	130	16

Table 1: **The test suite used to test the heuristics**

Program Name	Average number of nodes
dual	2.51
fft	30.95
fib	1.05
mm	9.68
pyramid	3.17
whet	7.01

Table 2: **Average number of nodes in the Ready List**

with a large number of data constraints, the improvement in the hit rate was minimal due to the lack of opportunity to reorder. The heuristical nature of the scheduler can be observed in the negative gains obtained for certain programs, such as `fib` with a cache size of 2 lines.

5 Conclusion

In this paper, we have presented a program representation, the *schedule graph*, that provides a framework for data cache optimization. We use the framework to prove that the problem of data cache miss minimization is NP-complete. We then use the framework to design a heuristic for generating good schedules that improve the performance of the data cache. Preliminary experiments with a prototype implementation of the heuristic indicate that program execution can be improved by instruction reordering to improve data cache performance.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1986.

Program	Number of cache misses									
	8 bytes		16 bytes		32 bytes		64 bytes		128 bytes	
	Org.	Reord.	Org.	Reord.	Org.	Reord.	Org.	Reord.	Org.	Reord.
dual	19	17	19	17	18	16	15	15	15	15
fft	98	59	76	50	62	43	34	34	32	32
fib	3	4	3	3	3	3	3	3	3	3
mm	89	44	51	12	7	7	7	7	7	7
pyramid	8	7	6	7	6	6	6	6	6	6
whet	14	15	4	4	4	4	4	4	4	4

Table 3: **Comparison of cache misses before and after reordering**

[2] E. G. Coffman, editor. *Computing and Job-Shop Scheduling Theory*, New York, London, Sydney, Toronto, 1976. John Wiley and Sons.

[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(5):587–616, October 1988.

[4] Michael R. Garey and David S. Johnson. *Computer And Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman And Company, Reading, Bell Laboratories, 1979.

[5] R. Gupta. Employing register channels for the exploitation of instruction level parallelism. *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[6] Y.-J. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. *Proceedings of Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 344–352, August 1991.

[7] B. A. Malloy, E. L. Lloyd, and M. L. Soffa. Scheduling dags for asynchronous multiprocessor execution. *IEEE Transactions on Parallel and Distributed Computing*, 5(5):498–508, May 1994.

[8] Scott McFarling. Program optimization for instruction caches. *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[9] Scott McFarling. Cache replacement with dynamic exclusion. *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 191–200, May 1992.

[10] Wen mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, 1989.

[11] Abraham Mendlson, Shlomit S. Pinter, and Ruth Shtokhamer. Compile time instruction cache optimizations. *Computer Architecture*, 12(3):177–886, July 1994.

[12] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[13] B. R. Rau. Data flow and dependence analysis for instruction level parallelism. *Languages and Compilers for Parallel Computing*, pages 237–251, Aug 1991.

[14] A. Dain Samples and Paul N. Hilfinger. Code reorganization for instruction caches. Technical report, Computer Science Division - EECS, University of California at Berkeley, October 1988.

[15] R. L. Sites. Instruction ordering for the cray-1 computer. Technical Report 78-CS-023, University of California at San Diego, July 1978.

[16] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[17] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.