Strategies for Cache and Local Memory Management by Global Program Transformation

By

Dennis Gannon, William Jalby and Kyle Gallivan

TECHNICAL REPORT NO. 228

Strategies for Cache and Local Memory Management By Global Program Transformation

by

Dennis Gannon, William Jalby and Kyle Gallivan

September, 1987

This report is based on work supported in part by the Air Force Office of Scientific Research under Grant No. AFOSR-86-0147.

Strategies for Cache and Local Memory Management by Global Program Transformation

Dennis Gannon

Dept. of Computer Science, Indiana University and
CSRD Univ. of Illinois, Urbana-Champaign.

William Jalby

INRIA and CSRD Univ. of Illinois, Urbana-Champaign

Kyle Gallivan

CSRD Univ. of Illinois, Urbana-Champaign.

ABSTRACT

In this paper we describe a method for using data dependence analysis to estimate cache and local memory demand in highly iterative scientific codes. The estimates take the form of a family of "reference" windows for each variable that reflects the current set of elements that should be kept in cache. It is shown that, in important special cases, we can estimate the size of the window and predict a lower bound on the number of cache hits. If the machine has local memory or cache that can be managed by the compiler, these estimates can be used to guide the management of this resource. It is also shown that these estimates can be used to guide program transformations in an attempt to optimize cache performance.

September 11, 1987

Strategies for Cache and Local Memory Management by Global Program Transformation

Dennis Gannon

Dept. of Computer Science, Indiana University and
CSRD Univ. of Illinois, Urbana-Champaign.

William Jalby

INRIA and CSRD Univ. of Illinois, Urbana-Champaign

Kyle Gallivan

CSRD Univ. of Illinois, Urbana-Champaign.

1. Introduction

Perhaps the most critical feature in the design of a shared memory parallel processor is the organization and the performance of the memory system. Generally, the shared memory is implemented as a set of independent modules (which, in turn, may be interleaved) connected to the processors via either a bus or a switch network. This organization exhibits several potential performance problems. First because of the long journey each memory reference must traverse in going to and from this shared resource the latency to access a data may be pretty high; this effect ends up to be very important in systems with large number of processors where the number of network stages that each memory request has to go through, grows as $\log_2(p)$ where p is number of processors (due to the number of stages of the network). Second due to the contention both at the network level and the memory level (routing and memory banks conflicts), the practical bandwidth (and also the latency) can be severely degraded. A good example of such phenomenon is "hot spot" contention [PhNo85]. To overcome these problems, the key idea is to use a hierarchical memory system, technique already proven efficient on sequential computers for speeding up memory access.

In such a system the memory is organized in several levels which might be fully shared (each processor may access the whole level), partially shared (the processors "inside a cluster" share the access to a given level) or fully private (each processor has its own level, access of which is restricted to itself; therefore access do not suffer from going through the communication medium and from interfering with the other processors). The transfers between these levels are either entirely hardware managed (such as with a cache where the user has no explicit control on the loading and unloading strategies) or fully software managed (such as registers where the user, or preferably, the compiler explicitly moves data between levels). For example the ALLIANT FX8 uses two levels of shared memory (trading size for speed): the main memory is connected via a bus to a high bandwidth cache which is turn shared by the processors through a crossbar. Access to vectors from the cache is 2 to 3 times faster than accessing from memory. Additionally, each processor has its own instruction cache and its own set of vector registers. In other machines only the main

memory level is fully shared and each processor has a private data cache, such as with the Sequent Balance and Encore Multimax, or a local memory which might be physically distinct from the main memory (CRAY2) or just a portion of the shared memory such as the IBM RP3 or BBN Butterfly. In this last case, the difference between local and shared requests are that the local ones do not go through the network. Finally, the Cedar system has perhaps one the most ambitious design of memory hierarchy. At the processor level, we have instruction caches and vector registers. The processors are then grouped into clusters which share a local cluster memory which is accessed by a shared cluster cache. Finally the clusters have access to the globally shared memory through the network. Additionally, each cluster has a prefetch unit controlled by software for prefetching data and therefore hiding the latency induced by the access to the global memory level.

However, the overall performance of all these hierarchical memory systems is highly dependent upon the address reference stream of the program (more precisely its locality). Several studies ([CJST85] [GaJaM87] [GaJa87]) have shown how algorithm reorganization may result in considerable performance improvement. It is crucial to notice that even in the case of hardware managed systems (ALLIANT FX8), reorganizing the program in order to make references to the same variable closer in time (or reducing the size of the working set) will speedup program execution [GaJaM87] [GaJa87]. Similar phenomena were already observed in studies considering the effect of program organization on paged memory behavior [KaMK69] and our approach can be considered following in the spirit of [AsKL81].

In this paper we consider the problem of automating the process of transforming programs to optimize the utilization of the memory hierarchy. For sake of simplicity, in our first approach we will assume that the transfers between levels are completely under software control. In that case, locality optimization is a 2-level process: first for a given program, one must solve the allocation problem (which data is to be kept in cache and for how long), and then restructure the program to minimize the number of data transfers. In fact, our basic strategy may still be applied to hardware managed systems. This is because reducing the number of transfers between levels is a dual problem to maximizing the reuse of data (optimizing "hit ratio"). Our key idea in solving the allocation problem is to consider it at a macroscopic level (loop level and section of arrays) rather at the microscopic level (machine level instruction and individual array elements). Using the fact that for scientific codes, most of the CPU time is spent in loop-like structure execution, we will globally study the interaction between two statements in a loop, by analyzing the sets of all the addresses referenced by each of them during the whole loop execution.

First we show that the theory of data dependence analysis used in automatic vectorizing compilers can be extended so that a more refined algebraic structure can be given to a class of data dependences associated with array index expressions that are common in scientific code. We call this class of dependences "uniformly generated". Next we associate a "reference window" with each data dependence. The reference window of a data dependence between two statements describes the set of elements (section of the array) that must be kept in the fast memory level to make sure that any data referenced by both statements will stay in the fast level as long as both statements continue to be executed and continue to reference that data item. More generally, we

try for a given dependence to determine the amount of space required to ensure that each data will be loaded just once. In fact, the reference window can be considered as the part of the "working set" (i.e. data which is going to be reused later and which may result in cache hits.) In sections 3 and 4 of the paper, we show that "uniformly generated" dependences have special properties that relate the structure of the data dependence graph to the lattice of reference windows and, given information about the loop bounds, we can estimate the size and "hit ratio" of the various windows. Now the problem is very similar to a classical bin packing problem: the size of the windows being the cost, and the "hit ratio" being the benefit associated with a window. In fact, all the elements necessary to manage the data between the different levels can be done symbolically at compile time, while the final decision might only be taken at runtime by substituting values in the symbolic expressions.

In section 5 of the paper, it is shown that program transformations like loop interchange and blocking can have a substantial effect on the size of the windows and therefore on the demand for space in the fast level. While this is a well known fact to most programmers, it is shown that the data dependence modeling can be used a mechanism to predict when a loop interchange can improve performance. In section 6 we show how this mechanism can be used to decide which data should be moved from the global memory of a multiprocessor system to the local memory. We also briefly discuss the implications for multiprocessors with shared cache.

2. Definitions

In this paper we use the standard definitions for data dependences given in many places (for a recent overview see [PaWo86]). A flow dependence from a statement S_1 to a statement S_2 exists when a value computed in S_1 is stored in a location associated with some variable name x which is later referenced and used in S_2 and is denoted

$$\delta_x : S_1 \rightarrow S_2$$

An anti-dependence from S_1 to S_2 exists when a variable x referenced by S_1 must be used before it is overwritten by S_2 and is denoted

$$\hat{\delta}_x : S_1 \rightarrow S_2$$

An output dependence from S_1 to S_2 exists when both statements modify a common variable x and S_1 must complete before S_2 does. This is denoted by

$$\delta_x^o : S_1 \rightarrow S_2$$

In order to track memory references another dependence type , known as an input dependence, is used. Unlike the other three types of dependences, an input dependence does not impose a constraint on the potential parallel execution of the two statements, but we still use a notation similar to the others:

$$\delta_x^I : S_1 \rightarrow S_2$$

In the case of references to elements of structured variables such as vectors or arrays, most references occur within loops. In this paper we consider only simple "for loop" iterations though much of what we say applies to "while loops" and other tail recursive control structures. For each data dependence between two references nested within a loop, we extend the work of [Cytron85] and associate with the dependence a set of **distance vectors** which is defined as follows. Consider a nested sequence of k loops of the form shown below.

```
For i_1 = L_1 to U_1
    For i_2 = L_2 to U_2
        .....
            For i_k = L_k to U_k
                ....
                S_1
                ....
                S_2
                ....
            endfor
        .....
    endfor
endfor
```

The module Z^k is called the **Extended Iteration Space** and the product $\prod_{i=1}^{k} D_i$ where D_i is the range of the i^{th} induction variable $[L_i \,..\, U_i]$, is called the **Bounded Iteration Space.** Both the extended and the bounded iteration spaces have a total order which is defined by the point in time at which the element is executed, i.e.

$$(v_1, v_2, ..., v_k) < (w_1, w_2, ..., w_k)$$

if there is a point s, $1 \leq s \leq k$, such that $v_i = w_i$ for $i < s$ and $v_s < w_s$. We will say a vector is **positive** if it is greater than zero in this order.

If a data dependence exists between a variable reference (which might be a component of a structured variable x) in S_1 at iteration

$$i_1{=}i_1^0\,,\; i_2{=}i_2^0\,,\; \cdots\; i_k{=}i_k^0$$

and the same variable (or component of x) in S_2 is referenced at iteration

$$i_1{=}i_1^1\,,\; i_2{=}i_2^1\,,\; \cdots\; i_k{=}i_k^1$$

Let

$$v_1{=}i_1^1{-}i_1^0\,,\; v_2{=}i_2^1{-}i_2^0\,,\; \cdots\quad v_k{=}i_k^1{-}i_k^0.$$

If this vector is positive in the total order of the iteration space, then we say the dependence exists and has a distance vector $\overline{V} = (v_1, v_2, ..., v_k)$ at time

$$\bar{I}^0 = (i_1^0, i_2^0, \ldots, i_k^0).$$

In general there may be more than one distance vector at each point in time, because S_2 may reference the same component of x at several different points in the future. To make this more precise, let d be the dimension of the structure x and let k be the depth of loop nesting that we are considering. Let S_1 reference x by an indexing function $f: Z^k \rightarrow Z^d$ and S_2 reference x by the function $g: Z^k \rightarrow Z^d$. Then the dependence

$$\delta_x: S_1(\ldots x[f(\bar{I}^0)]\ldots) \rightarrow S_2(\ldots x[g(\bar{I}^1)]\ldots)$$

defines a family of distance vectors at iteration time $\bar{I}$ by the relation

$$V_{\delta,\bar{I}} = (v \in Z^k \mid v > 0 \text{ and } f(\bar{I}) = g(\bar{I} + v)).$$

Note that we insist that the distance vectors point forward in time (which translates to the requirement that the leading non-zero component of the vector be positive.)

There are a number of important, very common special cases. We say the dependence is **Uniformly Generated** if there is a linear function $h: Z^k \rightarrow Z^d$ and two vectors C_f and C_g such that

$$f(\bar{I}) = h(\bar{I}) + C_f$$
$$g(\bar{I}) = h(\bar{I}) + C_g$$

In this case it is easy to see that

$$V_{\delta,\bar{I}} = (v \in R^k \mid v > 0,\ h(v) = C_f - C_g)$$

and that the right hand side is constant in time (independent of $\bar{I}$). Clearly if a non-negative member of $h^{-1}(C_f - C_g)$ does not exist the dependence does not exist. If $h^{-1}(C_f - C_g)$ is a single vector $(v_1, v_2, \ldots, v_k)$ then we say the dependence is **uniquely generated** and denote it by,

$$\delta_x(v_1, v_2, \ldots, v_k) : S_1 \rightarrow S_2$$

Another method of describing the set of distance vectors is to represent it as the sum of a uniquely defined positive vector plus the kernel of h. To do this let $v = (v_1, v_2, \ldots, v_k)$ be the smallest non-negative vector in $h^{-1}(C_f - C_g)$ then clearly

$$V_{\delta,\bar{I}} = (v + w \mid w \in Ker(h))$$

where $Ker(h)$ is the kernel of the mapping, i.e. the set of all w such that $h(w) = 0$. The existence and uniqueness of v is because the time order is total and the function h is continuous.

To illustrate these ideas consider the loop

```
For i = 1 to n
    For j = 1 to n
S1      x[i,j] = 13.5
S2      y[i] = x[i-3,j+5] + 19.0
S3      z[j] = 1.4
S4      call f(3,z[j-7])
```

```
        endfor
    endfor
```

This program has six dependences and we will look at three of them. On the variable x, there is a flow dependence for each (i,j) pair to iterates $(i+3,j-5)$ when $i<n-3$ and $j>5$. In this case $h(i,j) = (i,j)$ and $C_f = (0,0)$ and $C_g = (3,-5)$. Because the $Ker(h)$ is trivial we can write this as

$$\delta_x(3,-5) : S_1 \rightarrow S_2$$

For the variable y we note that for each j iteration $y[i]$ is modified. Thus for each j we have a family of output dependences one from each value of i to the next. In this case $h(i,j) = i$. This is a self dependence so we have $C_f = C_g$ and the set of distance vectors is just equal to $Ker(h)$ which is generated by the vector (0,1). This dependence vector is written as

$$\delta_y^o(1,0)^+ : S_2 \rightarrow S_2$$

Where the superscript "+" is used to denote that a full module of dependences are generated by this vector, i.e. we have $\delta_y(k,0)$ for all $k>1$ and $(k,0) = k(1,0)$. Such dependences are called **cyclic self-dependences** and are very important for cache management.

The third dependence involves the variable z. Here we have $h(i,j) = j$, $C_f = 0$ and $C_g = -7$. The kernel of h is generated by the vector (1,0) and $h(0,7) = C_f - C_g$. Hence the set of dependence vectors can be described as

$$\delta_z(\ (0,7) + (1,0)^+\ ) : S_3 \rightarrow S_4$$

where the vector summation is to denote the one parameter family of vectors

$$(0,7) + p^*(1,0) = (p,7) \quad \text{for } all \ \ p \in Z.$$

It is important to notice that for $p \geq 0$ this is a flow dependence, but for $p<0$ the direction in time is reversed and in fact we have described a set of antidependences

$$\hat{\delta}_z(-p,-7) : S_4 \rightarrow S_3 \quad all \ \ p<0$$

Of course not all data dependences are uniformly generated. For example,

```
    For i = 1, n
S1      x[2*i+3] = 29.9
S2      y[i] = x[4*i+7] - 39.0
    endfor
```

has an anti-dependence from S_2 to S_1 which, at iteration i, has a distance vector

$$\hat{\delta}_x(i+2) : S_1 \rightarrow S_2 \ \ 1 \leq i \leq (n-3)/2.$$

The vector is not uniformly generated because the vector is of the wrong form (it depends upon time) and the it is carried only by the even iterations. In this situation we use the classical

"direction" vector notation. In the case above the distance is always positive, so we denote it as $\hat{\delta}_x(+)$. If, on the other hand the lower bound of the for loop was -3, the distances would have ranged from -1 to $(n-3)/2$ and we use the notation $\hat{\delta}_x(-/+)$. If the lower bound of the loop were -2, the vector would be $\hat{\delta}_x(0/+)$, etc.

3. The Cache Window for a Dependence Vector

For our purposes we are interested in the following question:

Let two references to a variable be linked by a data dependence. Under what conditions can we be assured that if the first reference to the variable brings the current value into cache, then the second reference will result in a cache hit?

In general this is a very hard problem whose answer depends on more than a single data dependence. One must know not only more global program information, but also a great deal about the way the cache replacement policy works. As described in section 1, we take the approach that the compiler can restructure the program and suggest a replacement schedule that will be reasonably good. The consequence of letting the compiler manage the cache is that we may focus attention on one variable at a time and estimate the effect of program restructuring on the demands that are made on cache resources by that variable.

If the compiler has complete control of the cache and it is to be managed much like a massive set of registers, the basic cache optimization problem can be stated as follows: when we read a scalar or an element of a structured variable, should we, or should we not, keep the element in cache. If we have complete knowledge of the data dependence structure of the program, a reasonable solution is to see of that reference was the source (tail) of a data dependence. This means the element will be referenced again, if so, and if there is room, we should keep the element. To make this idea more precise, we need the following.

DEFINITION. The **reference window,** $W(\delta_X)_t$ for a dependence $\delta_X : S_1 \rightarrow S_2$ on a variable X at time t is defined to be to be the set all elements of X that are referenced by S_1 before t that are also referenced (according to the dependence) after t by S_2.

For example, the loop

```
   for i = 1, n
S1    x[i] = 1.4
S2    x[i-3] = y[i]
   end;
```

has an output dependence $\delta_x^o(3)$. If we set a break point at the top of iteration i we see

$$W(\delta_x^o(3))_{t=i} = (x[i-3], x[i-2], x[i-1])$$

In this case, we see that to make sure every reference to x in statment S_2 (excluding those few not referenced at all by S_1) is a cache hit we must be sure that for all $t=i$ the past four elements referenced by S_1 are kept in cache.

A more complex example is given by

```
   for i = 1, m
S1    y[i] = x[i]
      for j = 1, m
S2          x[j] = 1.5
   end;
```

In this case there are three families of dependences. First there is an anti dependence

$$\hat{\delta}_x(0/+)\colon S_1 \rightarrow S_2$$

which describes the use of $x[i]$ prior to the reassignment in S_2 for all iterations in the future. It is not hard to see that

$$W(\hat{\delta}_x(0/+))_{t=i} = (x[1], x[2], \cdots x[i-1])$$

Second, from S_2 to itself, we have

$$\delta_x^o(1,0)^+\colon S_2 \rightarrow S_2$$

which again represents a family of dependences $\delta_x(k,0)$ generated by the distance vector (1,0). In each case, at the top of the loop all m elements of x have been referenced and will be referenced again. This implies

$$W(\delta_x^o(1,0))_t = (x[1], \cdots x[m])$$

At the top of loop i, all future references to $x[k]$ for $k>i$ in statement S_1 have been previously generated by assignments in S_2. Consequently,

$$\delta_x(+)\colon S_2 \rightarrow S_1$$

has an associated window

$$W(\delta_x(+))_{t=i} = (x[i], x[i+1], \cdots x[m])$$

As the above example illustrates, different dependences generate different, but not necessarily disjoint, reference windows. In section 5 of this paper we will have need of a mechanism for picking families of dependence windows for cache management. Our problem here is to consider ways in which a "basis" of dependences can be chosen that somehow generates the entire family of reference windows for a given variable x so that we can measure the size of the total set of elements of x that must remain in cache. To do that we must first study a bit of the algebra relating the dependence graph to the family of reference windows.

DEFINITION. Suppose we have three dependences

$\delta_1\colon S_1 \rightarrow S_2$

$\delta_2\colon S_2 \rightarrow S_3$

$\delta_3\colon S_1 \rightarrow S_3$

with the property that if $x[\bar{i}]$ is referenced by S_1 and later by S_3 then it is referenced some time in between by S_2. In this case we say that $\delta_3 = \delta_1 + \delta_2$.

The relationship between the reference windows for these three dependences is given by the following result.

Lemma 3.1. Let $\delta_3 = \delta_1 + \delta_2$ define a relation between references in statements S_1, S_2 and S_3.

$$\delta_1 : S_1 \rightarrow S_2, \quad \delta_2 : S_2 \rightarrow S_3$$

We then have

$$W(\delta_1)_t \cap W(\delta_2)_t \subset W(\delta_3)_t \subset W(\delta_1)_t \cup W(\delta_2)_t$$

Furthermore, if $W(\delta_1)_t \cap W(\delta_2)_t$ is not empty there is a cyclic self-reference

$$\delta_0 : S_2 \rightarrow S_2$$

and

$$W(\delta_1)_t \cap W(\delta_2)_t \subset W(\delta_0)_t .$$

Proof. Let $x[\bar{i}]$ be an element of $W(\delta_3)_t$. This means that it is referenced at, or before time t by S_1 and later by S_3. Either it has already been referenced by S_2 which means that it is in $W(\delta_2)_t$ or it has not which means that it is in $W(\delta_1)_t$. This prove the right hand inclusion. To prove the left hand inclusion notice that if $x[\bar{i}]$ is an element of both $W(\delta_1)_t$ and $W(\delta_2)_t$ then it has been referenced by both S_1 and S_2 and will be referenced by S_3 in the future. This last fact puts it in $W(\delta_3)_t$.

Assume that $W(\delta_1)_t \cap W(\delta_2)_t$ is not empty. Let $x[i]$ be in the intersection. Then by δ_1 $x[i]$ is referenced by S_2 after t and because of δ_2 it is referenced by S_2 at or before t. This implies the existence of δ_0 and $x[i]$ is a member of $W(\delta_0)_t$.

In general one would like to find a family of dependences that are mutually disjoint and generate the entire set of dependence windows. In other words we would like the intersection term in the above expression to be empty and the second inclusion to be a set equality. Unfortunately, this is not always true. For example,

```
      for i = 1 to m
S1        x[i] = 1.5
          for j = 1 to m
S2            y[i,j] = x[j]
          end;
S3        z[i] = x[j-4]
      end;
```

has 6 dependences. Among these, three dependences satisfy the addition formula ($\delta_3 = \delta_1 + \delta_2$) and we have

$$W_1(\delta_x{:}S_1{\rightarrow}S_2)_{t=i} = (x[1], x[2], \ldots, x[i-1])$$

$$W_2(\delta_x{:}S_2{\rightarrow}S_3)_{t=i} = (x[i-4], x[i-3], \ldots, x[m])$$

$$W_3(\delta_x{:}S_1{\rightarrow}S_3)_{t=i} = (x[i-4], x[i-3], x[i-2], x[i-1])$$

which satisfies $W_3 = W_1 \cap W_2$ and not $W_3 = W_1 \cup W_2$. One of the reasons that this happens is that both the dependences $\delta_x{:}S_1{\rightarrow}S_2$ and $\delta_x{:}S_2{\rightarrow}S_2$ are not uniquely generated, i.e., they both have direction vectors $(-/+)$. If we restrict our attention to uniformly generated dependences it is possible to state a stronger results. In particular, we have

Theorem 3.1. Let S_1 reference $x[h(i)+C_f]$, S_2 reference $x[h(i)+C_g]$ and S_3 reference $x[h(i)+C_k]$. Then if $h()$ is linear and the dependences

$$\delta_1{:}S_1 \rightarrow S_2 \quad \text{and} \quad \delta_2{:}S_2 \rightarrow S_3$$

both exist, then a dependence $\delta_3{:}S_1 \rightarrow S_3$ exists and, in the unbounded iteration space, we have

$$W(\delta_3)_t = W(\delta_1)_t \cup W(\delta_2)_t$$

Proof. Let $x[i]$ be in $W(\delta_1)_t$ and let $v_i \in V_{\delta_i}$ for $i=1,2$ be the smallest positive vectors in these sets. By definition $x[i]$ is referenced before t by S_1 and after t by S_2. Looking at the reference by S_2 we have $i = h(\bar{i})+C_g$ with $\bar{i} > t$. Pick $\bar{i}$ to be the smallest such vector that satisfies this condition. Because $h(v_2) = C_k - C_g$, we have $\bar{i} + v_2 > t$ and $i = h(\bar{i}+v_2)+C_k$. Hence, $x[i]$ is referenced by S_3 after t as long as $\bar{i}+v_2$ lies in the iteration space. This is always true in the unbounded space, but in the bounded space it may not always hold.

In the other direction, let $x[i]$ be in $W(\delta_2)$ be referenced by S_2 at $\bar{i} < t$. If we pick $\bar{i}$ to be as large as possible, then there exists $v \in V_{\delta_2}$ such that $\bar{i}+v > t$ where S_3 references $x[i]$. But $\bar{i}-v_1 < t$ defines a point where S_1 references $x[i]$, so $x[i]$ is in $W(\delta_3)$. But again, in the bounded iteration space this may not hold.

This proves that, in the unbounded iteration space, we have

$$W(\delta_1) \cup W(\delta_2) \subset W(\delta_3)$$

To prove the inequality the other way, we need only observe that if $x[i]$ is in $W(\delta_3)$ and is referenced by S_1 at i_1 before t and at i_3 by S_3 after t. Consider $i_2 = i_3 - v_2$. If $i_2 > t$ then $x[i]$ is in $W(\delta_1)$, otherwise it is in $W(\delta_2)$.

DEFINITION. We say that a window for a dependence **spans** a set of other windows if each is contained in the first.

Theorem 3.1 states that for any three references generated by the same function $h()$ that form a graph connected by dependences, there is a single dependence that has a window that spans the others. The next step is to show that this may be generalized to larger sets.

DEFINITION. $UG(x)$ is defined to be the subset of the atomic data dependence graph consisting of nodes that are references to variable x and edges that are uniformly generated dependences.

We say that a *source* for a directed graph is a node where all non-self cycle edges are outgoing and a *sink* is an edge where all non-self cycle edges are ingoing.

Theorem 3.2. Let $C \subset UG(x)$ be a connected component. Then C has a spanning dependence and following three statements are true.

1. **If p is a source then it is unique and there is a node q and a dependence $\delta: p \rightarrow q$ such that $W(\delta)$ spans C.**
2. **If q is a sink then it is unique and there is a node p and a dependence $\delta: p \rightarrow q$ such that $W(\delta)$ spans C.**
3. **If there are no sinks or sources in C then the references windows for dependences in C are all equal.**

Proof. We first show that C is complete in the sense that if x and y are nodes in C then there is a dependence between them. We use an induction based on Theorem 3.1. Let p and q be two nodes in C. Pick the shortest (undirected) path between p and q. If this path has length greater than 1 then let r be the node on the path connected to p and s be the node connected to r. Nodes p, r and s form a triangle with two sides that are dependences. As in the proof of Theorem 3.1 the existence of the third edge (between p and s) follows from the linearity of $h()$. Hence there is a shorter path between p and q which contradicts the assumption that the minimum was greater than 1. The fact that the source and sink are unique follow from this completeness.

We next prove that if there is a source or sink then there is a spans dependence edge connected to it. Let si be a sink for the graph and let $\delta_1: p \rightarrow q$ be an arbitrary edge not involving si. By the closure property there are two edges $\delta_2: p \rightarrow si$ and $\delta_3: q \rightarrow si$. By Theorem 3.1 δ_2 spans both. By the same argument all edges connected to si are totally ordered by the set inclusion of the corresponding reference windows, hence there is a maximal element that spans all other edges connected to si and hence all edges in C. If a source node exists then the same argument shows that there is an edge connected to the source that spans all others. If both a source and a sink exist then the spanning edge must be connected to both. If neither exist, then let q be any node. Then because q is not a source or sink there must be both an ingoing edge $\delta_1: x \rightarrow q$ and an outgoing edge $\delta_2: q \rightarrow y$. Again by Theorem 3.1 both δ_1 and δ_2 are spanned by an edge from x to y. This fact must be true for any outgoing and ingoing pair. This means that x can not be an end point to an edge to any maximal window that strictly spans any other, hence all the reference windows must be equal.

It should be noted that this result tells us how to pick a dependence that spans any connected component of a dependence graph in an unbounded iteration space. Unfortunately, in the bounded iteration space, Theorem 3.1 is no longer valid, and remain true only 'away from the boundary' of the iteration space.

Fortunately there is another mechanism that we can use to estimate window extents in a bounded iteration space. The key idea is that for each window in a bounded iteration space we can enclose the window in a moving "frame" of fixed size. Let D^k be the bounded iteration space and

let x be a structured variable if dimension n. If we have a dependence

$$\delta_x\colon x[h(i) + C_1] \rightarrow x[h(i) + C_2]$$

we can define the following special subspaces of Q^k where Q is the field of rationals. Let e_i for $i = 1..k$ define the natural basis of Z^k that corresponds to the induction varibles $i_1, i_2, \ldots, i_k$. Define the subsets of Z^k,

$$V_j = span(e_j, e_{j+1}, \ldots, e_k)$$

where $span(u_1, \ldots, u_t)$ is defined as

$$[\, v \in Q^k \mid v = \sum_{r=1}^{t} \alpha_r u_r, \text{ with } \alpha_r \in Q \,]$$

We have

$$V_k \subset V_{k-1} \cdots \subset V_2 \subset V_1 = Q^k.$$

We can now characterize the window for δ_x as follows.

Theorem 3.3. Let r_1 be the largest integer such that $ker(h) \subset V_{r_1}$ and let $v \in h^{-1}(C_1 - C_2)$. Let r_2 be the index of the leading nonzero term in v and set $r = min(r_1, r_2)$. Define

$$X_v = <-s*v + V_{r+1}> \cap \overline{D}^k.$$

where $\overline{D}^k$ is the closure of the iteration space over the rationals and s is a rational in the range $[0, 1]$. We then have

$$W(\delta_x)_{t=(i_1,i_2,\ldots,i_k)} \subset (\, x[I] \mid I \in h(i_1, \ldots, i_r, 0, 0, \ldots, 0) + C_1 + h(X_v)\,)$$

Proof. Let $x[s_1] = x[s_2]$ be two references to the same element of x where $s_1 = h(t_1) + C_1$ and $s_2 = h(t_2) + C_2$ and $t_1 \leq t < t_2$. Because $s_1 = s_2$, we have

$$h(t_2 - t_1) = C_1 - C_2.$$

Letting $\bar{t} = t_2 - t_1$, we have $\bar{t} - v \in ker(h)$. Because $ker(h) \subset V_r$, we can let $h = \bar{t} - v$ which must take the form

$$h = (0, \ldots, 0, h_r, h_{r+1}, \ldots, h_k).$$

Now consider two case. First assume $t_1 > t - v$. We now have

$$t_1 \leq t < t_1 + v$$

Assume t_1 is of the form

$$t_1 = (\bar{i}_1, \bar{i}_2, \ldots, \bar{i}_k).$$

Because the first $r-1$ terms of v are zero we have

$$\bar{i}_j \leq i_j \leq \bar{i}_j + v_j$$

hence $\bar{i}_j = i_j$ for all $j < r$. Consequently, if we let

$$t_0 = (i_1, i_2 \ldots i_r, 0, 0, \ldots, 0)$$

then we can write

$$t_1 = t_0 - s^*v + w$$

where $s \leq 1$ is chosen to satisfy $\bar{i}_r = i_r - s^*v_r$ and w is of the form

$$(0, 0, 0, \ldots, w_{r+1}, w_{r+2}, \cdots w_k)$$

with $w_j = \bar{i}_j - i_j + s^*v_j$. Applying h() and adding C_1 we have

$$s1 = h(t_1) + C_1 = h(t_0) + C_1 + h(-s^*v+w)$$

which is of the correct form because $-s^*v+w \in X_v \cap \bar{D}^k$.

Now assume that $t_1 \leq t - v$. If $t_1+h < t-v$ we would have $t_2 = t_1+h+v < t$ which is wrong. Hence, $t_1 + h \geq t - v$. By applying an argument identical to the one above, we can pick a number $s \leq 1$ that gives us

$$t_1 + s^*h = t_0 - v + w$$

where $w_j = \bar{i}_j - i_j + s^*h_j$ for $j > r$ and 0 otherwise. Again $-v+w$ is in X_v and because $h \in ker(h)$ we have

$$h(t_1) + C_1 = h(t_1 + s^*h) + C_1$$

and the theorem is proved.

The advantage of the formulation in Theorem 3.3 is that the window has been enclosed in a moving frame. The term

$$h(\, i_1, i_2, \ldots, i_r, 0, 0, \cdots, 0)$$

describes how the frame moves in time. The term

$$C_1 + h(\, X_v) \quad with \quad X_v = <-s^*v+V_{r+1}, s\in[0,1]> \cap \bar{D}^k$$

is the time-independant "frame" for the window. Notice that $h(X_v)$ is independent of the choice of v because any other choice will differ from v by a member of $Ker(h)$. In the following section it will be shown that this formulation provides the necessary machinery to compute the size of cache windows.

4. Hit Ratios and Selecting Cache Windows.

Assume, for now, that we have a machine where the compiler can select which memory references to keep in cache. (For local memory this is always the case.) Clearly we would like to select those references belonging to reference windows associated with dependences that somehow generate a lot of cache hits. Our problem is twofold:

1. How do we compute the total size of a reference window and what is the cache hit ratio if we keep the entire window in cache?

2. How do we decide which windows to keep and which to disguard when the total is too big to fit in cache?

Another question that we would like to answer is if we have no control over the cache then can we estimate the hit ratios for hardware cache policies other than the one above? While the mechanisms described in this paper can be used to solve this problem, we will not consider it here.

Our basic cache scheduling algorithm will be as follows. First, at compile time, we select a set of dependence windows that we consider important. Our policy for cache replacement will be the following: we will read an element into the cache as soon as it enters the window and remove it from cache as soon as it leaves the window.

A simple way to estimate the total number of elements in cache for this policy is simply to sum the window sizes for each of the selected dependences. Unfortunately, as we have seen the relationship between cache window in a system of dependencies for a given variable can be rather complex. In particular, cache windows overlap and an element might be counted several times by this scheme. For example,

```
        for i = 1 to m
S1          x[i] = 1.5
            for j = 5 to 9
S2              y[i,j] = x[j]
            end
S3          z[i] = x[i-3]
        endfor
```

In this case there are 6 dependences listed below.

$$W_1(\delta_x(0/+):S_1 \rightarrow S_2)_{t=i} = (x[5], x[6], \ldots, x[min(i-1,9)])$$

$$W_2(\delta_x(+/-):S_2 \rightarrow S_3)_{t=(i,j)} = (x[max(5,i-3)], \ldots, x[9])$$

$$W_3(\delta_x(\ 3\ ):S_1 \rightarrow S_3)_{t=i} = ([x[i-3], x[i-2], x[i-1])$$

$$W_4(\delta_x(1,0)_*:S_2 \rightarrow S_2)_{t=(i,j)} = (x[5], \ldots, x[9])$$

$$W_5(\delta_x(0/+):S_3 \rightarrow S_2)_{t=i} = (x[5], \ldots, x[min(9,i-3)])$$

$$W_6(\delta_x(+/-):S_2 \rightarrow S_1)_{t=(i,j)} = (x[max(5,i)], \ldots, x[9])$$

Notice that, in fact, there are only two significant windows here. One is the window of size 3, W_3, and the other is the window of size 5, W_4. Each of these are based on uniquely generated dependences. The other windows are subsets of W_4. The correct cache policy for this program is to select W_3 and W_4 to be kept in cache. Also notice that W_3 sweeps over the entire x array while W_4 is constant in time (after $i = 1$). At some times they are disjoint, but at times they overlap and it is only during the period of overlap that the other dependences exist at all.

Because of the fact that these non-uniformly generated dependences have reference windows that tend to be either subsets of uniformly generated dependences or they have low hit ratios, we drop them from consideration for inclusion into the cache. (We will attempt to justify this restriction better in the next section.) For now our approach is based on the following strategy.

Let $UG(x)$ be the subgraph of the atomic data dependence graph consisting of those reference nodes involving the variable x and those edges corresponding to uniformly generated dependences. For each connected component $C \subset UG(x)$ we estimate the size and the "hit ratio" of the reference window for the spanning dominate dependence. If the total size is greater than the cache capacity, we attempt to restructure the program to reduce the size. If the program cannot be restructured but the cache can be managed by tagging the references that should be retained in cache, we attempt to solve the corresponding bin packing problem to select the reference windows that will give best performance.

Our nest task is to describe the machinery for the size and hit ratios of uniformly generated dependences. In some cases the task is easy.

For example, the loop

```
for i = 1 to n
      for j = 1 to m
S1       ... X[i,j] ..
S2       ... X[i-3, j+5] ...
      endfor
endfor
```

defines a dependence with distance vector $\delta_X(3,-5)$. At iteration (i_o,j_o) $X[i_o,j_o]$ is referenced. This element is not referenced again until iteration (i_o+3,j_o-5). Consequently, statement S_1 will have to access elements

```
                  X[i   ,j], X[i   ,j+1], ... , X[i   ,m],
X[i+1,1] ..       X[i+1,j], X[i+1,j+1], ... , X[i+1,m],
X[i+2,1] ..       X[i+2,j], X[i+2,j+1], ... , X[i+2,m],
X[i+3,1],..,X[i+3,j-5]
```

This set of elements defines the window $W(\delta_X)$ for this dependence at iteration $(i+3,j-5)$. Clearly any smaller set would not include reference $X[i,j]$ and would cause a cache miss for statement S_2.

In the more general case of a uniformly generated dependence δ from a term of the form $x[h(I)+C]$ we need to consider the formulation from Theorem 3.3. Let $\overline{I} = (i_1, ..., i_k)$. Let $v \in V_\delta$ and r be chosen according to the conditions of the theorem. We have

$$W(\delta_x)_{t=\overline{I}} \subset h(i_1, ..., i_r, 0, 0, ..., 0) + C + h(X_v)$$

where

$$X_v = < -s^*v + span(e_{r+1}, \ldots, e_k) >$$

and e_j is the vector describing the extent of the j^{th} induction variable. The problem is to compute the size of $h(X_v)$. There is a natural mapping to the quotient module

$$h: Z^k \rightarrow \frac{Z^k}{Ker(h)}$$

which identifies all point in iteration space that reference the same element of the variable x by mapping each index point i to its equivalence class $[i]\ mod\ h$. If we let X be the set of all distinct elements of x, then there is an injection

$$In: \frac{Z^k}{Ker(h)} \rightarrow X$$

given by $In([i]) = h(i)+C$. For a point t in iteration space, let t_r be the projection on the first r components. Clearly we have another formulation of Theorem 3.3.

$$W(\delta_x)_t \subset h(t_r)+In(h(X_v))$$

so

$$|W(\delta_x)_t| \leq |h(X_v)|$$

To compute the size of $h(X_v)$ we use the following result. Let U be a convex subset of Z^k. The linear function $h()$ is composed of component functions of the form

$$h_i(v_1, \ldots, v_k) = \sum_{j=1}^{k} h_{i,j} v_j.$$

Lemma 4.1. Let $h_{i_1}, \cdots h_{i_r}$ be a linearly independent set of components of $h()$. Let VP be the set of vertecies of a bounded convex subset U of Z^k and let

$$p_i = gcd(\ h_{i,j}\ \ j=1..k\ )$$

and

$$s_i = max(1, \max_{v,w \in VP}(, |h_i(v) - h_i(w)|))$$

An upper bound on the size of the set $h(U)$ is given by

$$|h(U)| \leq \frac{s_{i_1}}{p_{i_1}} \frac{s_{i_2}}{p_{i_2}} \cdots \frac{s_{i_r}}{p_{i_r}}$$

The proof is by induction on r. When $r=1$ the image of h is a scalar in the bounded interval

$$[\ \min_{v \in U} h(v),\ \max_{w \in U} h(w)\ ].$$

Because U is convex and h is linear, the size of this range is bounded by s and the maximum occurs at the vertecies of U. But the image of h can assume values only every p points so the

number of elements is bounded by $\frac{s}{p}$. In general, we observe that $\frac{s_{i_r}}{p_{i_r}}$ is the number of hyperplanes orthogonal to h_{i_r} that contain the image of U. Let V be one of the hyperplanes and consider the convex set $V \cap U$. By induction,

$$|h(V \cap U)| \leq \frac{s_{i_1}}{p_{i_1}} \frac{s_{i_2}}{p_{i_2}} \cdots \frac{s_{i_{r-1}}}{p_{i_{r-1}}}$$

and we have $|h(V)| \leq |h(V \cap U)| \frac{s_{i_r}}{p_{i_r}}$.

To apply this result observe that the determination of a maximal set of linearly independent elements is a standard linear algebra computation. The only messy part of the computation is the determination of of the index r and a vector in V_δ. The rest is given by

Lemma 4.2. Let δ be uniformly generated based at a reference of the form $h(I)+C$. Let $v \in V_\delta$ and let r be chosen as in Theorem 3.3. Assume the iteration space takes the form $D = \prod_{i=1}^{k} [0, d_i]$. We have

$$|W(\delta_z)| \leq |h(X_v)|$$

where the verticies of X_v in Lemma 4.1 may be taken to be

$$0, -v, \quad \text{and} \quad (0, \ldots, 0, d_j, 0, \ldots, 0) \quad \text{for} \quad j \geq r+1$$

As an example consider the following matrix multiplication routine.

```
for i = 0 to n1-1
    for j = 0 to n2-1
        for k = 0 to n3-1
            a[i,j] += b[i,k]*c[k,j];
        end;
    end;
end;
```

For each of the three variables the dependences are self cycles

$$\delta_a(0,0,1)^+, \quad \delta_b(0,1,0)^+, \quad \delta_c(1,0,0)^+$$

corresponding to the three functions

$$h^a(i,j,k) = (i,j) \quad r = 3$$

$$h^b(i,j,k) = (i,k) \quad r = 2$$

$$h^c(i,j,k) = (k,j). \quad r = 1$$

In each case the v may be taken to be zero because these are self cycles. The corresponding X sets are

$$X^a = ((0,0,0))$$

$$X^b = ((0,0,k) \mid 0 \leq k < n3)$$

$$X^c = ((0,j,k) \mid 0 \leq j < n2, 0 \leq k < n3).$$

In the first case we clearly have

$$|W(\delta_a)| = |([(0,0,0])| = 1$$

where the square brackets denote equivalence classes mod $h()$. In the second case we invoke Lemma 4.1 and 4.2.

Notice that for the dependence $\delta_b(0,1,0)^+$, the verticies of X^b are

$$(0,0,0) \text{and } (0,0,n_3).$$

The components of $h_b()$ are

$$h_{b,1}(i,j,k) = i \qquad h_{b,2}(i,j,k) = k$$

and from Lemma 4.2,

$$p_1 = p_2 = 1, \qquad s_1 = 1, \; s_2 = n_3.$$

We have,

$$|W(\delta_b)| \leq \frac{s_1}{p_1}\frac{s_2}{p_2} = n_3.$$

In the case of δ_c we have vertecies

$$(0,0,0), (0,n_2,0), (0,0,n_3).$$

Again p_i is always 1 and the component functions are

$$h_{b,1}(i,j,k) = j \qquad h_{b,2}(i,j,k) = k$$

We get

$$s_1 = n_2, \qquad s_2 = n_3$$

and this time,

$$|W(\delta_c)| \leq \frac{s_1}{p_1}\frac{s_2}{p_2} = n_2 n_3.$$

Notice that in the example above the function $h()$ was an injective map from X_v to $W(\delta)$. It is not hard to show when the dimension of the $Ker(h)$ is 1 then this is always true for a self-cycle.

In the case of loops that are not perfectly nested these formulas still may work. For example, in

```
For i = 1 to n
    For j = 1 to m
        x[i,j] = 15.4
```

```
        endfor
        for j = 1 to m
            y[i,j] = x[i-3, j+7]
        endfor
    endfor
```

the "official" distance vector only reflects the level of "common" nesting, i.e. $\delta_x(3)$. The formula give a window size of 3. However, we may artificially extend the vector to $\delta_x(3,-7)$ and apply the formula to get the correct window size of $3m-7$. Consequently the formula is still valid here, but one must take extreme care in making the extension. An alternative approach is to recognize when a vector operation is carried by a dependence and scale the formula above by the size of the vector. For example,

```
For i = 1 to n
    x[i, 1:m ] = 15.4
    y[i, 1:m ] = x [ i-3, 7:m+7]
endfor
```

is equivalent to the example above, but the "vector scaled" formula would give us an upper bound of $3m$.

DEFINITION. The **hit ratio** $hr(W(\delta_x))$ of a dependence window which is selected to be in cache is defined to be the number of times elements of $W(\delta_x)$ are referenced while in cache divided by the total number of times they are referenced.

A uniquely generated dependence $\delta_x(v):S_1 \rightarrow S_2$, if kept in cache, will have a hit ratio of at least $1/2$ because each element is referenced twice and the first reference is the read that loads the element in cache. If a dependence window is associated with a self-dependence $\delta_x(v)^+:S_1 \rightarrow S_1$ the ratio is much higher. In particular, we need to compute the number of times each element of the window is referenced. We introduce a new set which approximates $h^{-1}(W(\delta))$. Define the hull of a dependence δ at time t by

$$Hull(\delta)_{t=I} = (p \in D \mid I - v \leq p \leq I \quad \text{for } all\ v \in V_\delta).$$

The Hull is the set of all points p in iteration space that can reach to or beyond t by a vector from V_δ. Computing its size first involves computing its extent by finding the largest vector in V_δ that still lies in the iteration space. This vector spans a comvex domain that contains the Hull and we can use Lemma 4.1 to calculate the size.

For example,

```
for i = 0 to n-1
    for j = 0 to m-1
        y[i,j] = x[3*i - 2*j]
    end
```

end

has an dependence $\delta_x^I(2,3)^+$. Let $k = min(n/2,\ m/3)$. The hull of the dependence is the rectangle with lower left corner (0,0) and upper right corner $(2k, m)$. From Lemma 4.1 we have

$$|hull(\delta)| \ = \ min(mn,\ \frac{2m^2}{3}) \quad \text{and} \quad |W(\delta)| \leq 2m \ + \ min(2m, 3n)$$

Because each element is read into the window once and the remaining references are hits we have

$$hr(W(\delta)) \ = \ \frac{|Hull(\delta)| - |W(\delta)|}{|Hull(\delta)|}$$

which for $2m >> 3n$ is approximately $\frac{n-2}{n}$.

In general, let $C \subset UG(x)$ be a connected component. Define

$$Hull(C)_{t=I} \ = \ (\ p \in D \ \mid I - v \leq p \leq \text{ for } all\ v \in V_\delta \text{ for } all\ \delta \in C\ )$$

Lemma 4.3. Let $C \subset UG(x)$ be a connected component with n nodes. If $h(X_v)$ elements are always kept in cache we will have a hit ratio of

$$hr(h(Hull(C)) \ \geq \ \frac{n\ |Hull(C)| - |h(Hull(C))|}{n\ |Hull(C)|}$$

Proof. Each of the n nodes represents a source of references for the variable. There will be at least n other references to each element of $Hull(C)$. The total number of distinct elements of the array x that are referenced is just $|h(Hull(C))|$.

Notice that, in general, $Hull(C) \ = \ Hull(\delta)$ where δ is a spanning dependence for C in the unbounded iteration space. Where this fails to be true is when C is not closed under transitivity in the bounded iteration space because there will be no corresponding dependence. For example in the loop

```
for i = 1 to 100
      x[i] = 1.0;
      y[i] = x[i-60];
      z[i] = x[i-140];
end;
```

In this case, there are only two dependences one of distance 60 from S1 to S2 and one of distance 80 from S2 to S3. In this case we form the transitive closure of C and work with the hull of the spanning dependence, even though the dependence may not exist in the bounded case. It is not hard to show that in this case $Hull(C) \subset Hull(\delta)$. From this point on, we assume that the iteration space is large enough to make sure C is closed.

Given these tools we may now formulate the basic cache management algorithm as follows. Let B be a block of code, most likely a subroutine. Assume, for now, that B has no subroutine or function calls that require a substantial amount of cache activity. Also assume that B is one large

(possibly nested) loop. (If not, then apply the algorithm below to each subblock contained in a top-level loop in B).

```
initialize S = empty
for every variable x in B {
    let C_x^i for i = 1..n_x connected components of UG(x).
    let δ_x^i be the dominating dependence for C_x^i.
    let n_x^i be the number of nodes in C_x^i.

    for i = 1 to n_x do  S = S + {δ_x^i, n_x^i};
    }
set Cache = empty
set n = size-of-cache
while (n > 0){
    let (δ, n) be the pair in S with the maximum value of
        n |Hull(δ)| - |W(δ)|.
    if |W(δ)| ≤ n {
        Cache = Cache + W(δ)
        n = n - |W(δ)|
        }
    S = S - (δ, n)
    }
```

Elements are selected in order of decreasing value of the total number of hits generated by the component if it were kept in cache. It is easy to see that if you are given the choice of two windows to keep in cache, the one that generates the greatest total number of cache hits will yield the best overall memory access performance. Upon termination, *Cache* is the family of connected component that have the best memory reference performance.

The algorithm above is presented to give a simple selection method. It may be sub-optimal. The complete solution of this selection problem is NP-complete, but good approximate algorithms do exist (see [SaSa75]).

5. Program Transformation That Improve Data Locality.

In the previous section it was shown that if we consider the union of all cache windows for uniformly generated dependences associated with variables in a program segment we can estimate the size and hit ratio of the elements in cache in terms of the loop bounds. Because the estimates are only based on uniformly generated dependences the hit ratio estimates will be lower bounds. Because windows belonging to different generating functions may overlap, our size estimates will tend to be upperbounds. If the union is too big to fit in cache, we have given an algorithm to select a reasonable subset to keep.

On the other hand, if the union of the cache window sizes are too big for cache, then another alternative is to try to restructure the program to reduce the size of the cache requirement. The basic set of transformations to do this are well known. In their study of paged memory management, Abu-Sufa, Kuck and Lawrie showed that a good strategy is to attempt to split loop and then remerge them together so that loops tend to use fewer different variables and if a variable is used more than once its uses tend to be clustered together.

In the case of cache management these same techniques apply and, in addition, several other transformations can be used. For example, consider the following program segment.

```
      for i = 1 to n
            for j = 1 to m
S2                x[i,j] = y[j];
S3                z[i,j] = y[j-3];
            end;
      end;
```

The uniformly generated dependence

$$\delta_y^I((0,3)+(1,0)^+)\colon S_2 \rightarrow S_3$$

dominates the other two (self-cyclic) dependences. The window is

$$W(\delta_y^I) = (y[1], y[2], \ldots, y[m])$$

which has size m. By interchanging the loops (which does not change the meaning of the program) we have

```
      for j = 1 to m
            for i = 1 to n
S2                x[i,j] = y[j];
S3                z[i,j] = y[j-3];
            end;
      end;
```

and the dependence becomes

$$\delta_y^I((3,0)+(0,1)^+)\colon S_2 \rightarrow S_3$$

which still dominates the other two (self-cyclic) dependences. The window is now

$$W(\delta_y^I)_{t=(i,j)} = (y[j-3], y[j-2], y[j-1])$$

which has size 3. If m was too big to let the window fit in cache then 3 will probably be small enough. In addition, this transformation did not change the cache hit ratio. In fact we have

Theorem 5.1. Any correctness preserving program transformation will leave the cache hit ratio of the reference windows of data dependences unchanged.

Proof. By definition, the reference window of a dependence $\delta_x: S_1 \rightarrow S_2$ at time t is the set of elements of x that have been referenced by S_1 in the past that will be referenced by S_2 in the future. This means that if the window is kept in cache there will be one initial load and the remainder of the references will be hits. A program transformation will not change the nature or total number of references. The ratio is the the total number of times an element of the window is referenced while in cache divided by the total number of times it is referenced. In these terms, both the numerator and denominator are invariant with respect to program transformation.

Unfortunately, loop interchange is not always powerful enought to reduce the size of the cache demand to do the job. There is, however, another technique known as loop blocking that can have a significant effect on the size of the cache requirements. Consider a nested sequence of k loops

```
for i1 = 0 to n1
    for i2 = 0 to n2
      ...
        for ik = 0 to nk
            Body(i1, i2, ..ik)
        end;
    end;
end;
```

We say the r^{th} loop has been **blocked** if we apply the transformation which replaces the the loop

```
for ir = 0 to nr
    ...
    ...
end;
```

with

```
for jr = 0 to nr/dr
    for ir = dr*jr to dr*(jr+1)
        ...
        ...
    end;
end;
```

where the value d_r is called the blocking factor. This form of blocking is always legal in that it

does not effect the order of execution of the statements nested within. Furthermore, by itself, it does nothing to reduce the size of the cache windows. Its power comes when it is used in combination with loop interchange.

For example consider the following simple code segment.

```
for i = 1 to n
    for j = 1 to m
        for k = 1 to n
            a[i,k] += b[k]*c[j,k]
        end;
    end;
end;
```

In this case we have

$$|W(\delta_a)| = n$$

$$|W(\delta_b)| = n$$

$$|W(\delta_c)| = m*n.$$

Assume $n > m$ and mn is to big for the cache which is of size CS. We apply blocking to the "for k" loop and get

```
for i = 1 to n
    for j = 1 to m
        for r = 0 to n/d
            for k = r*d to (r+1)*d
                a[i,k] += b[k]*c[j,k]
            end;
        end;
    end;
end;
```

Now, interchanging loops to bring the "for r" loop to the outer most position and normalizing the "for k" loop, we have

```
for r = 0 to n/d
    for i = 1 to n
        for j = 1 to m
            for k = 1 to d
                a[i,k+r*d] += b[k+r*d]*c[j,k+r*d];
            end;
```

```
            end;
        end;
    end;
```

The reference windows now have sizes

$$|W(\delta_a)| = d$$

$$|W(\delta_b)| = d$$

$$|W(\delta_c)| = m*d$$

Clearly, in this case we can choose d small enough so that the total window size $(m+2)*d < CS$ and every element will need to move only once from memory to cache and every other access will be a hit.

Unfortunately, it is not always possible to restructure so that we have a perfect memory reference behavior. For example, in the case of matrix multiply we can use blocking on three levels in the original algorithm shown below.

```
for i = 1,n
    for j = 1, n
        for k = 1,n
            a[i,j] = a[i,j]+b[i,k]*c[k,j];
        end;
    end;
end;
```

The result takes the form

```
for r = 0 to n/d1
    for s = 0 to n/d2
        for t = 0 to n/d3
            for i = 1 to d1
                for j = 1 to d2
                    for k = 1 to d3
                        a[i+r*d1,j+s*d2] += b[i+r*d1,k+t*d3]*c[k+t*d3,j+s*d2];
                    end
                end
            end;
        end;
    end;
end;
```

and we have

$$|W(\delta_a)| = d1*d2$$

$$|W(\delta_b)| = d1*n$$

$$|W(\delta_c)| = n^2$$

The reader will find it easy to verify that no matter how we block and reorder the loops there is always a window of size n^2. What this implies is that if n^2 is greater than CS we can not expect to get perfect cache performance. The question then becomes, how good can the cache hit ratio be made if we keep a subset of the full reference window?

To answer this question we observe that the computation of a reference window is also a function of the scope of the execution we are considering. In the case above, it is easy to see why the window for the variable c is so big. The entire (n^2 element) c array is accessed for each iteration of the outermost loop. If we restrict our view to the inner most 5 loops we see that

$$|W(\delta_c)| = d2*d3$$

which reflects the fact that for each r iteration of the outer loop, the five inner loops make $d1*n^2$ references to a sequence of sub-block of size $d2$ by $d3$ of the c array. Hence for each r iteration we reload each sub-block and the total hit ratio is

$$1 - \frac{1}{d1}.$$

As the value of r is incremented a different sub-block is loaded and the hit ratio is as above. Another way to say this is if

$$d1*d2 + n*d1 + d2*d3 \leq CS$$

then each element of a and b are accessed from memory only once and referenced from cache $n-1$ times, and each element of c is read from memory $\frac{n}{d1}$ times and referenced from cache $n - \frac{n}{d1}$ times.

In the program above we restricted the context of the program execution to derive a smaller cache window. In terms of the data dependences in the program, observe that the family of data dependence vectors V_δ for the variable c is

$$V_\delta = (p*v_r + q*v_i \mid p, q \in Z^+)$$

where the generating vectors v_r and v_i are

$$v^r = (1,0,0,0,0,0) \quad \text{and} \quad v_i = (0,0,0,1,0,0)$$

By restricting the context we have simply discarded the dependences related to the loop variable r and the dependence family is generated by v_i alone. The hull of the associated reference window contains the extreme points (0,0,0,0,0,0) and $(0,0,0,d1,d2,d3)$. After applying Lemma 4.2 we have $|W(\delta_c(v_i))| = d2*d3$.

6. The Role of Concurrency.

In the case of programs executing on parallel processing systems most of the analysis above can be directly applied, but some care must be taken. There are two cases that are of interest to us. In the first case, we consider shared memory multiprocessors where each processor has either a local memory or cache for its own private use. Examples of this type of machine include the BBN Butterfly, the IBM RP3, the Encore Multimax and, at the level of processor clusters, the Illinois Cedar. The other case is where the processors share a cache. The prime example of this is the Alliant FX/8.

In these multiprocessor systems there are three basic forms of concurrency that are used in scientific computations. The two that are the most simple are parallelized loops and vectorization and the third is where the computation takes the form of a system user defined light-weight tasks.

Consider first the case of the multiprocessor with private local memory. In the light weight task model, tasks communicate through shared variables or message buffers. Because the code has already been partitioned into tasks, the analysis in the previous sections can be applied to optimize the performance of each separate processor. Because shared variables and shared message buffers can not be cached without potential coherency problems, there is little that can be done here.

A more interesting case is the parallelization of loops. Following Kennedy and Allen [Allen83] we give the following

Definition. Let a loop L_i be the i^{th} loop in a nested sequence and let v_i be the vector (0, 0, $\cdots$, 1, 0, ..., 0) with a 1 in the i^{th} position and zero everywhere else. We say that the loop **carries no dependence** if v_i is orthogonal to V_δ for every flow, output, or anti-dependence δ between terms nested in L_i.

The basic theorem that permits the parallel execution of a loop is then stated as:

Theorem 6.1. A loop may be parallelized without the use of special synchronization if and only if the loop carries no dependence.

Proof: See [Allen83], [KKLPW81] or [Wolfe82].

The key point of this result is that all data dependences (except for input dependences) are resolved withing the body of the loop. Consequently each iteration can be executed on a separate processor. Any input dependences that may span the loop can be discarded, because the corresponding window would span two different caches or local memories. The resulting reference windows associated with the reduced graph tell us precisely the information we need to keep in the cache or local memory of each processor.

For example, consider the blocked loop in the previous section:

```
for r = 0 to n/d
    for i = 1 to n
        for j = 1 to m
```

```
                for k = 1 to d
                    a[i,k+r*d] += b[k+r*d]*c[j,k+r*d]
                end;
            end;
        end;
    end;
```

The outermost loop does not carry any dependences and hence, may be parallelized.

```
FORALL r = 0 to n/d
    for i = 1 to n
        for j = 1 to m
            for k = 1 to d
                a[i,k+r*d] += b[k+r*d]*c[j,k+r*d]
            end;
        end;
    end;
ENDFORALL;
```

The semantics of the FORALL is that each processor will take a single unique r iterate and execute it to completion and then take another until all iterations have been complete. For a fixed value of r, the three cache windows are (in terms of the local time for each processor as)

$$W(\delta_a)_{t=(i,j,k)} = a[i,\ 1+r*d\ ..\ (r+1)*d]$$

$$W(\delta_b)_{t=(i,j,k)} = b[1+r*d\ ..\ (r+1)*d]$$

$$W(\delta_c)_{t=(i,j,k)} = c[1..m,\ 1+r*d\ ..\ (r+1)*d]$$

The way in which this information is used depends on the machine. For example, on the BBN Butterfly a local memory reference can be three times faster than a remote reference. (If a number of processors try to access the same element then this may be much worse.) Furthermore the microcode provides a block transfer mechanism that can copy a vector from global memory to a local are at very high speed. Hence for the Butterfly, we would be best to use this information to compile in special code that prefetches the cache window with block transfers as follows

```
blocal[1..d]      = block_transfer(b[1+r*d..(r+1)*d]);
clocal[1..m, 1..d] = block_transfer(c[1..m, 1+r*d..(r+1)*d]);
for i = 1 to n
    alocal[1..d] = block_transfer(a[i, 1+r*d..(r+1)*d]);
    for j = 1 to m
        for k = 1 to d
            alocal[k] += blocal[k]*clocal[j,k];
```

```
            end;
        end;
        a[i, 1+r*d..(r+1)*d] = block_transfer(alocal[1..d]);
    end;
```

Notice that because the "for r" loop was perfectly parallelizable there was no need to be concerned about coherence problems. Also the form of the data dependences were enough to detect where the copies should be made and when an update was needed. Notice that there are only a few simple constraints that must be satisfied. First to parallelize the "for r" loop we would like to pick d small enough so the n/d is greater than or equal to the number of processors. Also, we need d small enough so that $(m+2)d \leq CS$ where CS is the local memory size on each processor. On the other hand the size of the task for each processor grows with d, and we would wish to pick d large enough so that the task size make the parallel loop overhead look small. Fortunately, on the Butterfly this overhead is relatively small.

In case a loop carries dependencies, we may still parallelize the program, but we must synchronize the processors so that the dependence constraint is satisfied. Such a loop schedule is often called a "do-across" schedule. For systems that have private local memories for each processor, it is not likely that we would chose to cache data that is involved in a data dependence constraint. This is because such a constraint requires that one processor write data to a shared location before another processor reads it, or it must read/write the data before another processor overwrites it. In any event, the dependencies associated with "synchronized" data must not be included in our computation of cache needs because it must be shared between processors. After excluding these dependencies, what we are left with are the dependencies associated with data that is not involved in a synchronization conflict. It is purely local and we can apply the theory above to estimate its size and hit ratios.

In the case of a system where the cache is shared by all processors, such as the Alliant FX/8, the situation is vastly different from the private memory case. In particular, in the private cache case, processors are best working on data that is completely disjoint form the other processors. On the other hand, if the cache is shared, processors will be best off when they share the data in cache.

6. Conclusions.

The ideas presented here represent only a first approximation to the problem of transforming programs to improve cache behavior. Many important refinements are needed. In particular, this paper only considers the special case of uniformly generated dependences. It is shown that, in this case, we may make a good approximation to the size of the cache windows if certain conditions are satisfied. The problems come in two places. First, the structure of the bounded iteration space can make it very hard to select a set of spanning dependences that really reflect the lifetime of a variable in cache. In particular, if we generate a spanning dependence by a transitive closure operation, the dependence will exist in the unbounded iteration space but it may not exist in the bounded case. If it does not exist in a given iteration space, but we use it to compute a cache

window, we may overestimate the real cache demand. In general, this happens when the distance vectors are greater than the size of the iteration space.

The second shortcoming of the theory above is that if the dependences are not uniformly generated then the reference window for the dependence will, in general, not be of constant size. This can make it very hard to make a'priori choices of the best dependences to use for loading the cache. It should be possible to extend the theory so that we can detect upper bounds on reference windows for these non-uniformly generated dependences. These extensions should improve the range of programs where good estimates can be made.

Most of the theory in this paper assumes a cache that is completely under the control of the program. Even in the case of caches that are not completely managed by the user, the analysis developed in this paper may be usefull. It is easy to imagine a cache which is partially programmable by the user, namely one where cache loading is user programmed while the unloading is under control of the hardware. For example moving data from main memory can be performed by 2 kinds of instructions: fetch and store in the cache (the data is fetched from main memory and is stored in the processor registers as well as in the cache), fetch only (the data is fetched from main memory and stored in the processor registers but not kept in cache). The main advantage this policy is to avoid filling the cache (and therefore discarding usefull data) with useless data. The main difficulty lies in deciding at compile time by the restructurer what kind of move to perform. The key idea in managing such a cache is to force data that the analysis has decided should not be kept in cache to be fetched by fetch only instructions. The remaining problem is that we are not sure that the hardware unloading policy will follow our optimal programmed policy. However, if the hardware policy is based on an LRU scheme, it should be possible to analyze the behavior of the cache using our method because our static analysis generates a "worst case" stream of references and therefore we should be able to build a timetable indicating when the elements are touched. Using this information we will be able model the LRU policy and derive a good replacement scheme.

Currently, we are in the process of implementing the ideas described in this paper into a FORTRAN restructuring system being built at Indiana University and CSRD in Urbana. The objective is to show that an interactive program restructurer can take data provided by the program and the programmer and then make estimates about potential cache performance problems and also make suggestions about way that the programmer can restructure the code to avoid these problems. A detailed description of the software system and the results of experimental use will be given in a sequel to this paper.

7. References

[Allen83] J.R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph.D. Thesis, Rice University, Houston, Texas, April 1983.

[AlKe84] J. Allen, and K. Kennedy, "A Parallel Programming Environment," technical report, Rice COMP TR84-3, July 1984.

[ASKL79] W. Abu-Sufah, D. Kuck and D. Lawrie, "Automatic Program Transformations for Virtual Memory Computers," Proc. of the 1979 Nat'l Computer Conf., June, 1979, 969-974.

[ASKL81] W. Abu-Sufah, D. Kuck and D. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Program Transformation," IEEE Trans. Comp. V. C30 no. 5, May, 1981, pages 341-356.

[BuCy86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," Proc. Sigplan 86 Symposium on Compiler Construction, 21(7):162-175, July 1986.

[CoMK69] Coffman, E., McKeller, "The organization of matricies and matrix operations in the paged multiprogramming environment," CACM V. 12 pp. 153-165. March 1969.

[CGST85] W. Crowther, J. Goodhue, E. Starr, R Thomas, W. Milliken, T. Blackadar, "Performance Measurements on a 128-node Butterfly Parallel Processor," Proceedings of 1985 International Conference on Parallel Processing, pp. 531-540, 1985.

[Cytron84] R. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, University of Illinois, Urbana-Champaign Aug., 1984 Report No. UIUCDCS-R-84-1177).

[FeOt83] J. Ferante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10208, Aug. 1983.

[GaJa87] Gannon, D., Jalby, W., "The Influence of Memory Heirarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor," to appear in "The Characteristics of Parallel Algorithms," Gannon, Jamieson, Douglas, eds, MIT Press, 1987.

[Kenn80] K. Kennedy, "Automatic translation of Fortran programs to vector form," Rice Technical Report 476-029-4, Rice University, October 1980

[KKLW80] D. Kuck, R. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," IEEE Computer Society, proc. of the 4th Inter"l Computer Software and App. Conf., October, 1980, 709-715.

[KKLPW81]D. J. Kuck, R. H. Kuhn, B. Leasure, D. H. Padua and M. Wolfe, "Dependence graphs and compiler optimizations," Conference Record of Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, VA., January 1981.

[KuWM84] D. Kuck, M. Wolfe, and J. McGraw, "A Debate: Retire FORTRAN?," Physics Today, May, 1984, 67-75.

[Padua79] D. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois, Urbana-Champaign, Nov. 1979.

[PaKu80] D. Padua and D. Kuck, "High-Speed Multiprocessors and Compilation Techniques," IEEE Transactions on Computers, Vol. C-29, No. 9, September, 1980, 763-776.

[PaWo86] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," CACM, 29(12):1184-1201, Dec. 1986.

[PhNo85] G. Phister, A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," Proceeding of the 1985 International Conference on Parallel Processing, 1985, 790-797.

[Poly86] C. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. Thesis, University of Illinois Center for Supercomputer Research and Development. CSRD TR.595, Aug. 1986.

[SaSa75] Saraj, Sahni, "Approximate Algorithms for the 0,1 Knapsack Problem," JACM V. 22 no. 1 pp 115-124.

[KWDG87] Wang, K.-Y., Gannon, D., "Applying AI Techniques to Program Optimization for Parallel Computers," To appear in "AI Machines and Supercomputer Systems", Hwang, DeGroot, eds. McGraw Hill, 1987.

[Wolfe82] M. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Urbana-Chanpaign, 1982, Report no. UIUCDCS-R-82-1105.