LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-481

AUTOMATIC PARTITIONING OF PARALLEL LOOPS FOR CACHE-COHERENT MULTIPROCESSORS

Anant Agarwal
David Kranz
Venkat Natarajan

December 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors

Anant Agarwal, David Kranz
Laboratory for Computer Science, NE43-624
Massachusetts Institute of Technology
Cambridge, MA 02139
Phone: (617) 253-1448
email: agarwal@mit.edu
Venkat Natarajan
Motorola Cambridge Research Center
Cambridge, MA 02139

Abstract

This paper presents a theoretical framework for automatically partitioning parallel loops to minimize cache coherency traffic on shared-memory multiprocessors. The framework introduces the notion of uniformly intersecting references to capture temporal locality in array references, and the idea of data footprints to estimate the communication traffic between processors. The framework uses lattice theory to compute the size of data footprints. We demonstrate that algorithms based on our framework discover optimal partitions in many cases, such as non-communication-free parallelogram partitions of affine loop index functions, which were not handled by previous algorithms. We also show that our framework correctly reproduces results from previous loop partitioning algorithms proposed by Abraham and Hudak and by Sadayappan and Ramanujam. Because they deal only with index expressions, the algorithms are computationally efficient as well. We have implemented a subset of this framework for rectangular partitioning in a compiler for the cache-coherent Alewife machine.

1 Introduction

Cache-based multiprocessors are attractive because they seem to allow the programmer to ignore the issues of data partitioning and placement. Because caches dynamically copy data close to where it is needed, repeat references to the same piece of data do not require communication over the network, and hence reduce the need for careful data layout. However, the performance of cache-coherent systems is heavily predicated on the degree of temporal locality in the access patterns of the processor. Loop partitioning for cache-coherent multiprocessors is an effort to increase the percentage of references that hit in the cache.

The degree of reuse of data, or conversely, the volume of communication of data, depends both on the algorithm and on the partitioning of work among the processors. (In fact, partitioning of the computation is often considered to be a facet of an algorithm.) For example, it is well known that a matrix multiply computation distributed to the processors by square blocks has a much higher degree of reuse than the matrix multiply distributed by rows or columns.

Loop partitioning can be done by the programmer, by the run time system, or by the compiler. Relegating the partitioning task to the programmer flies in the face of the central

rationale for building cache-coherent shared-memory systems. While partitioning can be done at run time (for example, see [1, 2]), it is hard for the run time system to optimize for cache locality because much of the information required to compute communication patterns is either unavailable at run time or expensive to obtain. Thus compile-time partitioning of parallel loops is important.

This paper focuses on the following problem in the context of cache-coherent multiprocessors. Given a program consisting of parallel do loops (of the form shown in Figure 1 in Section 2.1), how do we derive the tile shapes of the iteration-space partitions to minimize the communication traffic between processors.

1.1 Contributions of This Work

This paper develops a theoretical framework for loop and data partitioning for cache-coherent distributed-memory multiprocessors. While the main focus of this paper is loop partitioning, the framework unifies the analysis required for either loop or data partitioning, or both. We indicate the modifications necessary for data partitioning. The loop partitioning specifies the shape of iteration space tiles that minimizes communication traffic in cache-coherent multiprocessors. The framework allows the partitioning of doall loops where the index expressions in array accesses can be any affine function of the indices.

Our analysis uses the notion of uniformly intersecting references to categorize the references within a loop into classes that will yield cache locality. The notion of data footprints is introduced to capture the combined set of data accesses made by references within each uniformly intersecting class. Then, an algorithm to compute the total size of the data footprint for a given loop partition is presented. While general optimization methods can be applied to minimize the size of the data footprint and derive the corresponding loop partitions, we demonstrate several important special cases where the optimization problem is very simple.

The size of data footprints can also be used to guide program transformations to achieve better cache performance in uniprocessors as well. Ferrante et al. [5] also address this problem and we discuss their work in Section 5.

We show that Abraham and Hudak's results on rectangular loop partitioning for caches [6] (where the index expressions are of the form index variable plus a constant) are reproduced by our theoretical framework. The framework, however, is able to handle much more general index expressions, and produce parallelogram partitions if desired.

We also show that the framework correctly produces the communication-free loop partitions for the class of programs handled by Ramanujam and Sadayappan [7]. In addition, the same framework is able to discover optimal partitions in cases where communication free partitions are not possible – a case not handled by [7].

A subset of the framework, including both loop and data partitioning has been implemented in the compiler system for the Alewife multiprocessor. The implementation currently handles rectangular partitions only. We are currently working on implementing the general framework.

1.2 Overview of the Paper

The rest of this paper is structured as follows. Section 2 states our system model and our program-level assumptions. Section 3 first presents a few examples to illustrate the basic ideas

```
Doall (i1, l1, u1)
  Doall (i2, l2, u2)
    ...
    Doall (im, lm, um)
      loop body
    EndDoall
    ...
  EndDoall
EndDoall
```

Figure 1: Structure of a single loop nest

behind loop partitioning; it then develops the theoretical framework for partitioning and presents several additional examples. The implementation of our compiler system and a sampling of results is presented in Section 4, and Section 7 concludes the paper.

2 The Problem Domain and Assumptions

We address the problem of partitioning loops in cache-coherent shared-memory multiprocessors. Partitioning involves deciding which loop iterations will run collectively in a thread of computation. Computing loop partitions involves finding the set of iterations which when run in parallel minimizes the volume of communication generated in the system. This section describes the types of programs currently handled by our framework and the structure of the system assumed by our analysis.

2.1 Program Assumptions

Figure 1 shows the structure of the most general single loop nest that we consider in this paper. The statements in the *loop body* have array references of the form $A(\vec{g}(i_1, i_2, \ldots, i_l))$, where the index function is $\vec{g} : \mathcal{Z}^l \rightarrow \mathcal{Z}^d$, l is the loop nesting and d is the dimension of the array A.

We address the problem of loop and data partitioning for index expressions which are affine functions of loop indices. In other words, the index function can be expressed as,

$$\vec{g}(\vec{i}) \quad = \quad \vec{i}\mathbf{G} + \vec{a} \tag{1}$$

where $\mathbf{G}$ is a $l \times d$ matrix with integer entries and $\vec{a}$ is an integer constant vector of length d.[1] We often refer to an array reference by the pair $(\mathbf{G}, \vec{a})$. (An example of this function is presented in Section 3.1). All our vectors and matrices have integer entries unless stated otherwise. We assume that the loop bounds are such that the iteration space is rectangular. However, we note that our methods can still be used to derive reasonable partitions when this condition is not met. Loop indices are assumed to take all integer values between their lower and upper bounds, i.e, the strides are one.

[1] Note that $\vec{i}$, $\vec{g}(\vec{i})$, and $\vec{a}$ are row vectors.

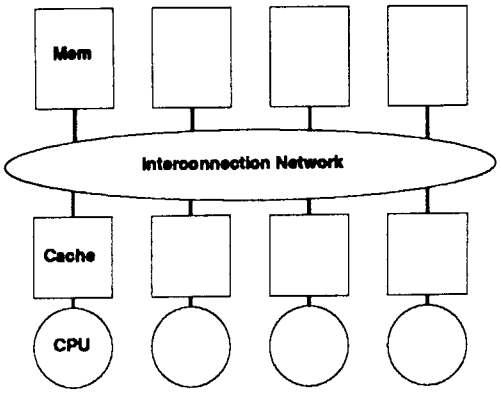

Figure 2: A system with caches and uniform access main memory.

Previous work [6, 7, 8] in this area restricted the arrays in the *loop body* to be of dimension exactly equal to the loop nesting. Abraham and Hudak [6] further restrict the *loop body* to contain only references to a single array; furthermore, all references are restricted to be of the form $A(i_1 + a_1, i_2 + a_2, \ldots, i_d + a_d)$ where a_j is an integer constant. Matrix multiplication is a simple example that does not fit these restrictions.

Given P processors, the problem of loop partitioning is to divide the iteration space into P tiles such that the total communication traffic on the network is minimized with the additional constraint that the tiles are of equal size, except at the boundaries of the iteration space. The constraint of equal size partitions is imposed to achieve load balancing. We restrict our discussions to parallelepiped tiles with emphasis on rectangular tiles. Rectangular tiles (of various aspect ratios) are important because it is easy to produce efficient code when the tile boundaries are simple expressions.

Like [6, 7, 8], we do not include the effects of synchronization in our framework. Synchronization is handled separately to ensure correct behavior. For example, in the doall loop in Figure 1, one might introduce a barrier synchronization after the loop nest if so desired. We also note that in some cases fine-grain data-level synchronization can be represented within a parallel do loop and its cost approximately modeled as slightly more expensive communication than usual [9]. See Apendix A for some details.

2.2 System Model

Our analysis applies to systems whose structure is similar to that shown in Figure 2. The system comprises a set of processors, each with a coherent cache. Cache misses are satisfied by global memory accessed over an interconnection network or a bus. The memory can be implemented as a single monolithic module (as is commonly done in bus-based multiprocessors), or in a distributed fashion as shown in the figure. The memory modules might also be implemented on the processing nodes themselves (data partitioning for locality makes sense only for this case). In all cases, our analysis assumes that the cost of a main memory access is much higher than a cache access, and that the cost of the main memory access is the same no matter where in main memory the data is located.

The goal of loop partitioning is to minimize the total number of main memory accesses. For simplicity, we assume that the caches are large enough to hold all the data required by a loop partition, and that there are no conflicts in the caches. When caches are small, the optimal loop

partition aspect ratios do not change, rather, the size of each loop tile executed at any given time on the processor must be adjusted so that the data fits in the cache. We assume that cache lines are of unit length. The effect of larger cache lines can be included as suggested in [6].

3 Loop Partitions and Footprints

This section develops a framework for compile time analysis of loops for performing optimal loop partitioning for a cache-coherent multiprocessor. After presenting two illustrative examples, we introduce the notions of a loop partition and the *footprint* of a loop partition with respect to a data reference in the loop. The footprints specify the data elements referenced within a loop partition. We then present the concept of uniformly intersecting references and a method of computing the cumulative footprint for a set of uniformly intersecting references. We develop a formalism for computing the volume of communication on the interconnection network of a multiprocessor for a given loop partition, and show how loop tiles can be chosen to minimize this traffic. The cumulative footprint can also be used to derive optimal data partitions for multicomputers with local memory.

3.1 Examples

This section presents examples to illustrate some of our definitions and to motivate the benefits of optimizing the aspect ratios of loop tiles. As mentioned previously, we deal with index expressions that are affine functions of loop indices. In other words, the index function can be expressed as,

$$\vec{g}(\vec{i}) \;=\; \vec{i}\mathbf{G} + \vec{a}$$

where $\mathbf{G}$ is a $l \times d$ matrix with integer entries and $\vec{a}$ is an integer constant vector of length d, termed the offset vector. Consider the following example to illustrate the above expression of index functions.

Example 1 *The reference $A(i_3 + 2, 5, i_2 - 1, 4)$ in a triply nested loop can be expressed by*

$$(i_1, i_2, i_3) \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} + (2, 5, -1, 4)$$

In this example, the second and fourth column of $\mathbf{G}$ are zero indicating that the second and fourth subscript of the reference is independent of the loop indexes. In such cases, we show in Section 3.4.1 that we can ignore those columns and treat the referenced array as an array of lower dimension. In future, without loss of generality, we assume that the $\mathbf{G}$ matrix contains no zero columns.

Now, let us introduce the concept of a loop partition by examining the following simple example.

Example 2

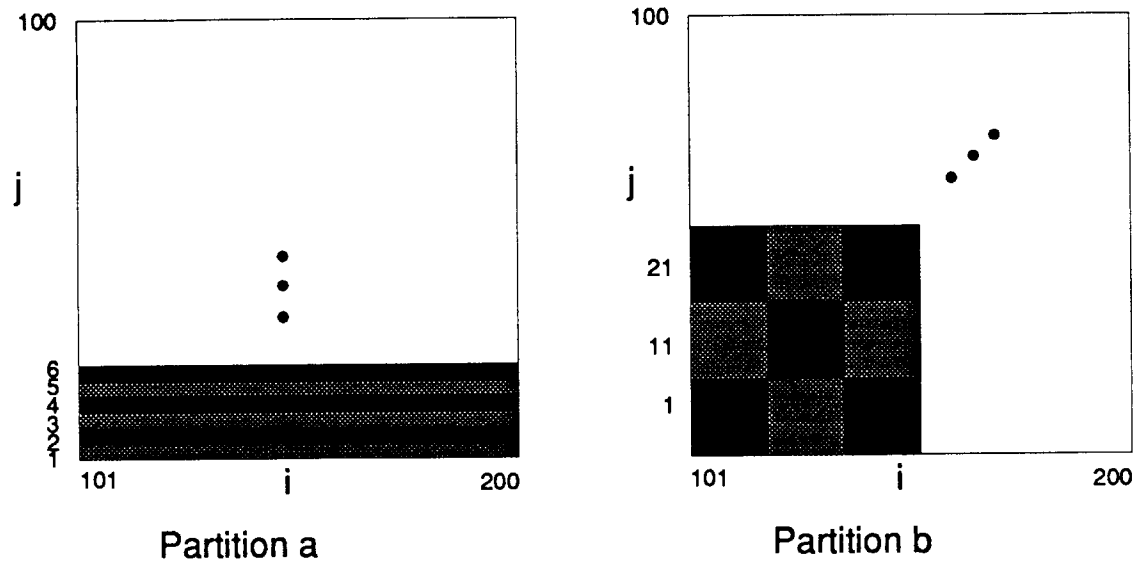

Figure 3: Two simple rectangular loop partitions in the iteration space.

Doall *(i, 101, 200)*
  Doall *(j, 1, 100)*
    A[i,j] = B[i+j,i-j-1]+B[i+j+4,i-j+3]
  EndDoall
EndDoall

Let us assume that we have 100 processors and we want to distribute the work among them. There are 10,000 points in the iteration space and so one can allocate 100 of these to each of the processors to distribute the load uniformly. Figure 3 shows two simple ways of partitioning the iteration space into 100 equal tiles.

Let us compare the two partitions in the context of a system with caches and uniform access memory by computing the number of cache misses. For each tile in partition **a**, the number of cache misses can be shown to be 104 whereas the number of cache misses in each tile of partition **b** can be shown to be 140. Although not obvious from the source code, partition **a** is a better choice if our goal is to minimize the cost of memory accesses. In fact, partition **a** has zero coherence traffic. Ramanujam and Sadayappan [7] presented algorithms to derive such coherence-traffic-free partitions when possible. In addition to producing the same partitions when coherence-traffic-free partitions exist, our analysis will discover partitions that minimize traffic when such partitions are non-existent as well (see Example 10).

Example 3

Doall *(i, 1, N)*
  Doall *(j, 1, N)*
    A[i,j] = B[i,j]+B[i+1,j+3]
  EndDoall

EndDoall

For Example 3, parallelogram tiles result in a lower cost of memory access compared to any rectangular partition since most of the inter iteration communication is internalized to within a processor. Although it is harder to perform load balancing and produce code in the case of parallelogram partitions, we would like to include them in the formulation of the loop partitioning problem. In higher dimensions a parallelogram tile generalizes to a hyperparallelepiped; the next subsection defines it precisely.

3.2 Loop Tiles in the Iteration Space

Loop partitioning results in a tiling of the iteration space. Loop partitioning is sometimes termed iteration space partitioning. A specific hyperparallelepiped loop tile is defined by a set of bounding hyperplanes.

Definition 1 *Given a l dimensional loop nest $\vec{i}$, each tile of a hyperparallelepiped* loop partition *is defined by the hyperplanes given by the rows of the $l \times l$ matrix $\mathbf{H}$ and the column vectors $\vec{\gamma}$ and $\vec{\lambda}$ as follows. The parallel hyperplanes are $\vec{h}_j\vec{i} = \gamma_j$ and $\vec{h}_j\vec{i} = \gamma_j + \lambda_j$, for $1 \leq j \leq l$. An iteration belongs to this tile if it is on or inside the hyperparallelepiped.*

We consider only hyperparallelepiped partitions in this paper; rectangular partitions are special cases of these. Furthermore, we focus on loop partitioning where the tiles are homogeneous except at the boundaries of the iteration space. Under these conditions of homogeneous tiling, the partitioning is completely defined by specifying the tile at the origin, namely $(\mathbf{H}, \vec{0}, \vec{\lambda})$, as indicated in Figure 4; the rest of this paper will deal with producing the aspect ratio for this tile. For notational convenience, we denote the tile at the origin as $\mathbf{L}$. More precisely, we use the following representation for the tile at the origin.

Definition 2 *Given the tile $(\mathbf{H}, \vec{0}, \vec{\lambda})$ at the origin of hyperparallelepiped partition, let $\mathbf{L} = \mathbf{L}(\mathbf{H}) = \Lambda(\mathbf{H}^{-1})^t$, where Λ is a diagonal matrix with $\Lambda_{ii} = \lambda_i$. We refer to the tile by the $\mathbf{L}$ matrix, as $\mathbf{L}$ completely defines the tile at the origin. $\mathbf{L}$ specifies the vertices of the tile at the origin. Often we also refer to the partition by the $\mathbf{L}$ matrix since each of the other tiles is a translation of the tile at the origin.*

Example 4 *For a rectangular partition, $\mathbf{H}$ is the identity matrix $\mathbf{I}$ and $\mathbf{L} = \Lambda$.*

3.3 Footprints in the Data Space

For a system with caches and uniform access memory, the problem of loop partitioning is to find an optimal matrix $\mathbf{L}$ that minimizes the number of cache misses. The first step is to derive an expression for the number of cache misses for a given tile $\mathbf{L}$. Because the number of cache misses is related to the number of unique data elements accessed, we introduce the notion of a *footprint* that defines the data elements accessed by a tile. The footprints are regions of the *data space* accessed by a loop tile.

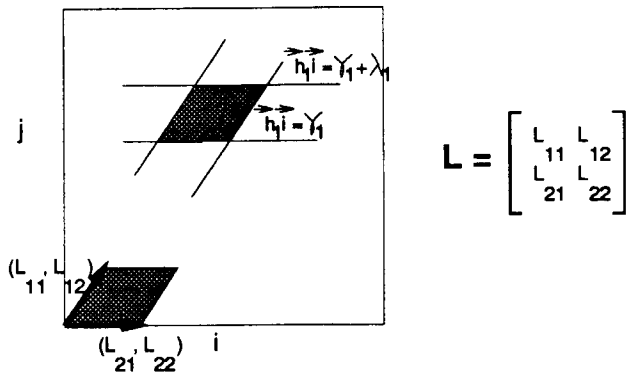

Figure 4: Iteration space partitioning is completely specified by the tile at the origin.

Definition 3 *The* **footprint** *of a tile* $(\mathbf{H}, \vec{\gamma}, \vec{\lambda})$ *of a loop partition with respect to a reference* $A[\vec{g}(\vec{i})]$ *is the set of all data elements* $A[\vec{g}(\vec{i})]$ *of* A*, for* $\vec{i}$ *an element of the tile.*

The footprint gives us all the data elements accessed through a particular reference from within a tile of a loop partition. Because we consider homogeneous loop tiles, the number of data elements accessed is the same for each loop tile.

We will compute the number of cache misses for the system with caches and uniform access memory to illustrate the use of footprints. The body of the loop may contain references to several variables and we assume that aliasing has been resolved; two references with distinct names do not refer to the same location. Let $A_1, A_2, \ldots, A_K$ be references to array A within the loop body, and let $F(A_i)$ be the *footprint* of the loop tile at the origin with respect to the reference A_i and let $F(A) = \bigcup_{i=1,\ldots,K} F(A_i)$ be the *cumulative footprint* of the tile at the origin. The number of cache misses with respect to the array A is $|F(A)|$. Thus, computing the size of the individual footprints and the size of their union is an important part of the loop partitioning problem.

To facilitate computing the size of the union of the footprints we divide the references into multiple disjoint sets. If two footprints are disjoint or mostly disjoint then the corresponding references are placed in different sets, and size of the union of their footprints is simply the sum of the sizes of the two footprints.

However, references whose footprints overlap substantially are placed in the same set. The notion of *uniformly intersecting references* is introduced to specify precisely the idea of "substantial overlap". Overlap produces temporal locality in cache accesses, and computing the size of the union of their footprints is more complicated.

The notion of uniformly intersecting references is derived from definitions of intersecting references and uniformly generated references.

Definition 4 *Two references* $A[\vec{g}_1(\vec{i})]$ *and* $A[\vec{g}_2(\vec{i})]$ *are said to be* intersecting *if there are two integer vectors* $\vec{i}_1, \vec{i}_2$ *such that* $\vec{g}_1(\vec{i}_1) = \vec{g}_2(\vec{i}_2)$*. For example,* $A(i+c1, j+c2)$ *and* $A(j+c3, i+c4)$ *are intersecting, whereas* $A[2i]$ *and* $A[2i+1]$ *are non-intersecting.*

Definition 5 *Two references* $A[\vec{g}_1(\vec{i})]$ *and* $A[\vec{g}_2(\vec{i})]$ *are said to be* uniformly generated *if*

$$g_1(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_1 \text{ and } g_2(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_2$$

where G is a linear transformation and $\vec{a_1}$ and $\vec{a_2}$ are integer constants.

The concept of uniformly generated references has been used in earlier work in the context of *reuse* and iteration space tiling [4, 3]. The intersection of footprints of two references which are not uniformly generated is often very small. So the total communication generated by two non-uniformly intersecting references is essentially the sum of the individual footprints.

However, the condition that two references are uniformly generated is not sufficient for two references to be intersecting. As a simple example, $A[2i]$ and $A[2i+1]$ are uniformly generated, but the footprints of the two references do not intersect. For the purpose of locality optimization through loop partitioning, our definition of reuse of array references will combine the concept of uniformly generated arrays and the notion of intersecting array references. This notion is similar to the equivalence classes within uniformly generated references defined in [4].

Definition 6 *Two array references are* uniformly intersecting *if they are both intersecting and uniformly generated.*

Example 5 *References $A[i,j], A[i+1, j-3], A[i, j+4]$ are uniformly intersecting, while the references $A[i,j], A[2i,j]$ are not. For more examples, see Appendix B.*

Footprints in the data space for a set of uniformly intersecting references are translations of one another, as shown below. The translation offset vector is related to the offset vector $\vec{a}_r$ of the reference $r = (G, \vec{a}_r)$.

Proposition 1 *Given a loop tile at the origin $\mathbf{L}$ and references $r = (G, \vec{a}_r)$ and $s = (G, \vec{a}_s)$ belonging to a uniformly generated set defined by $\mathbf{G}$, let $F(r)$ denote the footprint of $\mathbf{L}$ with respect to r, and let $F(s)$ denote the footprint of $\mathbf{L}$ with respect to s. Then $F(s)$ is simply a translation of $F(r)$, where each point of $F(s)$ is a translation of a corresponding point of $F(r)$ by an amount given by the vector $(\vec{a}_s - \vec{a}_r)$. In other words,*

$$F(s) = F(r) \vec{+} (\vec{a}_s - \vec{a}_r).$$

This follows directly from the definition of uniformly intersecting references. Recall that an element i of the loop tile is mapped by the reference $(G, \vec{a}_r)$ to data element $\vec{d}_r = \vec{i}\mathbf{G} + \vec{a}_r$, and by the reference $(G, \vec{a}_s)$ to data element $\vec{d}_s = \vec{i}\mathbf{G} + \vec{a}_s$. The translation vector, $(\vec{d}_s - \vec{d}_r)$, is clearly independent of i.

Although, the number of iteration points in two different tiles may not be identical for parallelogram partitions, they are equal from a practical viewpoint. Further, for rectangular partitions the tiles are identical except for translation. Also, from a practical viewpoint the size of the footprint is the same for all the loop tiles except for the ones at the boundaries. So we can focus on the loop tile at the origin.

Proposition 2 *The number of iterations in the tile (or the* volume *of the tile) $(\mathbf{H}, \vec{\gamma}, \vec{\lambda})$, is approximately the sum of the volume of the tile and one half the number of iteration points on the boundaries of the tile. The volume of the tile is $|\det \mathbf{L}(\mathbf{H})|$, which is the same as the volume $|\det \mathbf{L}|$ of the tile $\mathbf{L}$ at the origin.*

Proposition 3 *The number of iterations in a rectangular tile $(\mathbf{I}, \vec{\gamma}, \vec{\lambda})$ is $\prod_{i=1}^{l}(\lambda_i + 1)$.*

The volume of cache traffic imposed on the network is related to the size of the cumulative footprint. We describe how to compute the size of the cumulative footprint in the following two sections as outlined below.

- First, we discuss how the size of *the footprint for a single reference* within a loop tile can be computed. In general, the size of the footprint with respect to a given reference is not the same as the number of points in the iteration space tile.
- Second, we describe how the size of *the cumulative footprint for a set of uniformly intersecting references* can be computed. The sizes of the cumulative footprints for each of these sets are then summed to produce the size of the cumulative footprint for the loop tile.

3.4 Size of a Footprint for a Single Reference

This section shows how to compute the size of the footprint (with respect to a given reference and a given loop tile **L**) efficiently under certain conditions on **G**. Although we believe that these conditions cover a majority of practical cases, we summarize in Section 3.8 how to extend the techniques presented in this section with weaker conditions on **G**. We begin with a simple example to illustrate our approach.

Example 6

Doall *(i, 0, 99)*
  Doall *(j, 0, 99)*
    A[i,j] = B[i+j,j]+B[i+j+1,j+2]
  EndDoall
EndDoall

Let us suppose that the loop tile at the origin **L** is given by

$$\begin{bmatrix} L_1 & L_1 \\ L_2 & 0 \end{bmatrix}.$$

Figure 5 shows this tile at the origin of the iteration space. The reference matrix **G** is

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

The footprint of the tile at the origin with respect to the reference $B[i+j, j]$ is shown in Figure 6. The matrix

$$\mathbf{F}(B[i+j,j]) = \mathbf{LG} = \begin{bmatrix} 2L_1 & L_1 \\ L_2 & 0 \end{bmatrix}$$

describes the footprint. The integer points on or inside the parallelogram specified by **LG** is the footprint of the tile. So the size of the footprint is $|\det(\mathbf{LG})|$ plus the number of integer points on the boundary of the parallelogram. This computation reduces to $L_1L_2 + L_1 + L_2$. In

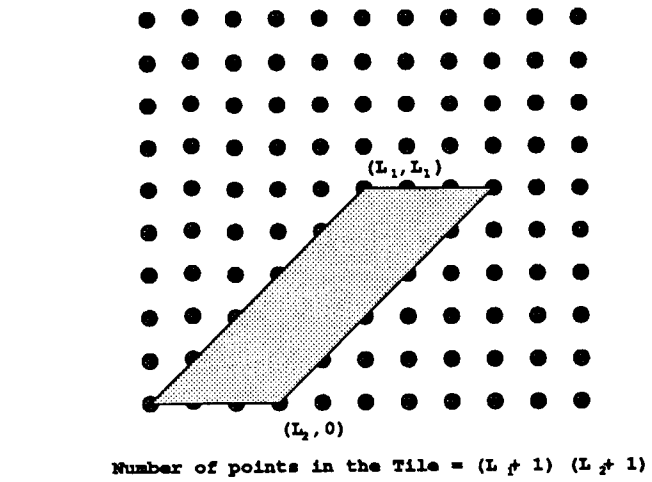

Figure 5: Tile **L** at the origin of the iteration space.

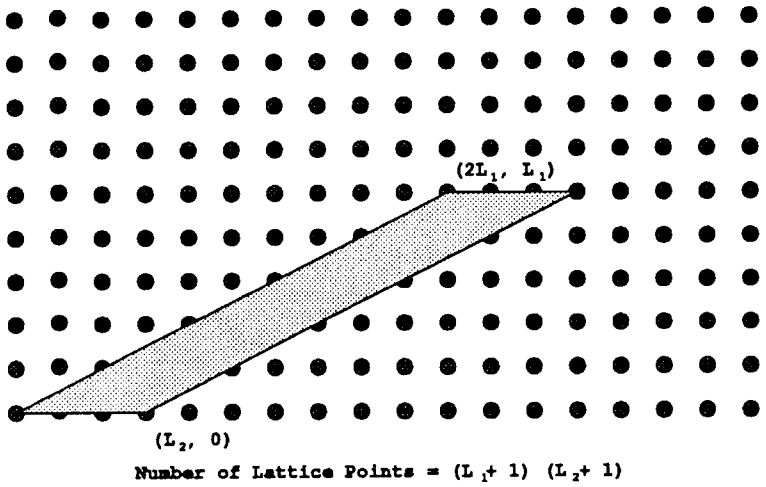

Figure 6: Footprint of **L** wrt $B[i+j, j]$ in the data space.

the rest of our discussion, for brevity, we will drop explicit mention of the integer points on the boundary of the parallelograms and use

$$\textit{Size of the footprint defined by } \mathbf{LG} = |\det(\mathbf{LG})| \tag{2}$$

The above example leads to the following questions. In general, is the footprint exactly the integer points on or inside **LG**? If not, how do we compute the footprint? The first question can be expanded into the following two questions.

- Is there a point in the footprint that lies outside the hyperparallelepiped **LG**? It follows easily from linear algebra that it is not the case.
- Is every integer point inside of **LG** an element of the footprint? It is easy to show this is not true and a simple example corresponds to the reference $A[2i]$.

We first study the simple case when the hyperparallelepiped **LG** completely defines the footprint. Precise definition of the set $S(\mathbf{LG})$ of points defined by the matrix **LG** is as follows.

Definition 7 *Given a matrix* $\mathbf{Q}$ *whose rows are the vectors* $\vec{q_i}$, $S(\mathbf{Q})$ *is defined as the set*

$$\{\vec{x} = a_1\vec{q_1} + a_2\vec{q_2} + \ldots + a_m\vec{q_m} | 0 \leq a_i \leq 1\}.$$

$S(\mathbf{Q})$ *defines all the points on or inside the hyperparallelepiped defined by* $\mathbf{Q}$.

So for the case where $\mathbf{LG}$ completely defines the footprint, the footprint is exactly the integer points in $S(\mathbf{LG})$. One of the cases where $\mathbf{LG}$ completely defines the footprint, is when $\mathbf{G}$ is unimodular as shown below.

Lemma 1 *The mapping of the iteration space to the data space as defined by* $\mathbf{G}$ *is one to one if and only if the rows of* $\mathbf{G}$ *are independent.*

Proof: $\vec{i_1}\mathbf{G} = \vec{i_2}\mathbf{G}$ *implies* $\vec{i_1} = \vec{i_2}$ *if and only if the rows of* $\mathbf{G}$ *are linearly independent.*

Lemma 2 *The mapping of the iteration space to the data space as defined by* $\mathbf{G}$ *is onto if and only if the columns of* $\mathbf{G}$ *are independent and the* g.c.d. *of the subdeterminants of order equal to the number of columns is 1.*

Proof: *Follows from the Hermite normal form theorem [10].*

Theorem 1 *The footprint of the tile defined by* $\mathbf{L}$ *with respect to the reference* $\mathbf{G}$ *is identical to the integer points on or inside the hyperparallelepiped* $\mathbf{LG}$ *if* $\mathbf{G}$ *is* unimodular.

Proof: *Immediate from the above two lemmas.*

We make the following two observations about Theorem 1.

- $\mathbf{G}$ is unimodular is a sufficient condition; but not necessary. An example corresponds to the reference $A[i + j]$. Further discussions on this theorem is beyond the scope of this paper and will be dealt with in a separate paper.
- One may wonder why $\mathbf{G}$ being onto is not sufficient for $\mathbf{LG}$ to coincide with the footprint. Even when every integer point in $\mathbf{LG}$ has an inverse, it is possible that the inverse is outside of $\mathbf{L}$. The one to one property of $\mathbf{G}$ guarantees that no point from outside of $\mathbf{L}$ can be mapped to inside of $\mathbf{LG}$. The reason for this is the one to one property is true even when $\mathbf{G}$ is treated as a function on reals.

It is also possible to apply Theorem 1 to compute the size of a footprint when the columns of $\mathbf{G}$ are not independent, as shown below.

3.4.1 Footprint Size when Columns of G are Dependent

We derive a $\mathbf{G'}$ from $\mathbf{G}$ by choosing a maximal set of independent columns from $\mathbf{G}$, such that $\mathbf{G'}$ is unimodular. If a unimodular $\mathbf{G'}$ exists, then it completely specifies the footprint, because all the points within the hyperparallelepiped defined by $\mathbf{LG'}$ will be accessed. We can now apply Theorem 1 to $\mathbf{LG'}$ to compute the size of the footprint.

Example 7 *Consider the reference $A[i, 2i, i+j]$ in a doubly nested loop. The columns of the* **G** *matrix*

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

are not independent. We choose $\mathbf{G}'$ *to be*

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Now $\mathbf{LG}'$ *completely specifies the footprint.*

It is possible that none of the maximal independent columns satisfy the conditions in Theorem 1. Such cases are dealt with in Section 3.8.

3.5 Size of the Cumulative Footprint for a Loop Tile

The size of the cumulative footprint for a loop tile is computed by summing the sizes of the cumulative footprints for each of the sets of uniformly intersecting references. First, let us present a method for computing the size of the cumulative footprint for a set of uniformly intersecting references.

The complexity of the computation of the size of the cumulative footprint for a set of uniformly intersecting references depends on the reference matrix **G** that defines the set and the loop partition **L**. We first focus on the types of references for which the conditions stated in Theorem 1 are true. In other words, we assume that **G** is unimodular.

Let us start by illustrating the computation of the cumulative footprint for Example 6.

The references to array B form a uniformly intersecting set and are defined by the following **G** matrix.

$$\mathbf{G} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Let us suppose that the loop partition **L** is given by

$$\begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}.$$

Then **LG** is given by

$$\begin{bmatrix} L_{11}+L_{12} & L_{12} \\ L_{21}+L_{22} & L_{22} \end{bmatrix}.$$

$ABCD$ and $EFGH$ shown in Figure 7 are the footprints of the tile **L** with respect to the two references ($B[i+j, j]$ and $B[i+j+1, j+2]$ respectively) to array B. In the figure, $\vec{AB} = (L_{11}+L_{12}, L_{12})$, $\vec{AD} = (L_{21}+L_{22}, L_{22})$, and $\vec{AE} = (1, 2)$.

The size of the cumulative footprint is the size of footprint $ABCD$ plus the number of data elements in $EPDS$ plus the number of data elements in $SRGH$. We can approximate the number of data elements by the area $ABCD + SRGH + EPDS$. Ignoring, the areas of the two triangles APE and HSD, we can approximate the total area by

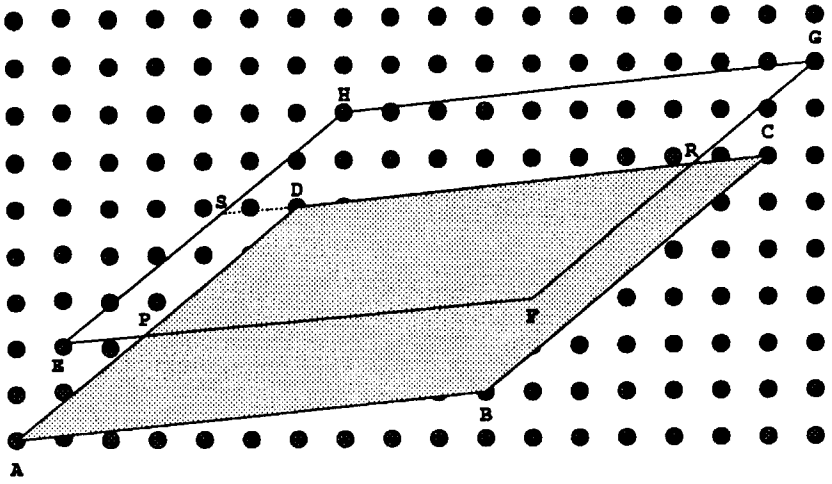

Figure 7: Data footprint wrt $B[i+j, j]$ and $B[i+j+1, j+2]$

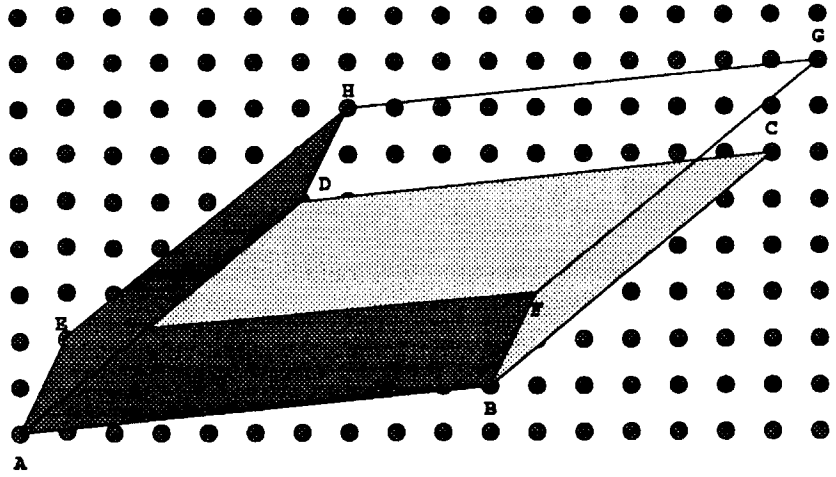

Figure 8: Difference between the cumulative footprint and the footprint.

$$\left|\det\begin{bmatrix} L_{11}+L_{12} & L_{12} \\ L_{21}+L_{22} & L_{22} \end{bmatrix}\right| + \left|\det\begin{bmatrix} L_{11}+L_{12} & L_{12} \\ 1 & 2 \end{bmatrix}\right| + \left|\det\begin{bmatrix} 1 & 2 \\ L_{21}+L_{22} & L_{22} \end{bmatrix}\right|$$

This approximation is reasonable if we assume that the constant terms in a uniformly intersecting set of references are small compared to the tile size. The first term in the above equation represents the area of the footprint of a single reference, i.e., $|\det(\mathbf{LG})|$. The second and third terms are the determinants of the $\mathbf{LG}$ matrix in which one row is replaced by the offset vector $\vec{a} = (1, 2)$. Figure 8 is a pictorial representation of the approximation.

We can generalize this idea of computing the cumulative footprint from multiple overlapping footprints when there are many references in a set of uniformly intersecting references as follows. Let the *spread* of the offset vectors $\vec{a}_i$ (or the constant terms) among a set of uniformly intersecting references be captured by a vector $\hat{a}$ as defined here. The vector $\hat{a}$ is derived from the set of offset vectors for the given set of uniformly intersecting references. Recall, that the offset vector $\vec{a}$ is a term in the index function as given in Equation 1.

Let the index vector of reference r from a set of uniformly intersecting references be $\vec{g}(\vec{i}) = \vec{i}\mathbf{G} + \vec{a}_r$, where $\vec{i}$ corresponds to the iteration space (as defined in Equation 1). The index vector has length d, where as before d is the dimension of the array.

Definition 8 *Define $\hat{a}$ such that the k^{th} component of $\hat{a}$ is the difference between the maximum and the minimum of the k^{th} component of all the references in the given uniformly intersecting set. In other words, $\hat{a}$ is a vector of length d where*

$$\hat{a}_k = \max_r(a_{r,k}) - \min_r(a_{r,k}), \forall k \in 1, \ldots, d.$$

$\hat{a}$ is referred to as the spread *of the uniformly intersecting set of references.*

For caches, we use the $max - min$ formulation (or the spread) to calculate the amount of communication traffic because the data space points corresponding to the footprints of references whose offset vectors have values somewhere between the max and the min lie within the cumulative footprint calculated using the spread.[2]

Theorem 2 *Given a hyperparallelepiped tile $\mathbf{L}$ and a reference matrix $\mathbf{G}$ which is unimodular, the size of the cumulative footprint with respect to a set of uniformly intersecting references specified by the reference matrix $\mathbf{G}$ and a spread vector $\hat{a}$ is approximately*

$$|\det \mathbf{LG}| + \sum_{i=1}^{d} |\det \mathbf{LG}_{i \to \hat{a}}|$$

where $\mathbf{LG}_{i \to \hat{a}}$ is the matrix obtained by replacing the ith row of $\mathbf{LG}$ by $\hat{a}$.

As before, we can deal with the case when the columns of $\mathbf{G}$ are not independent by choosing a maximal set of independent columns. We also have sharper estimates on communication volume when the tile is rectangular. Furthermore, we can also weaken the conditions on $\mathbf{G}$ when the tile is rectangular as shown in Section 3.7.

Finally, as stated earlier, the total communication generated by non-uniformly intersecting sets of references is essentially the sum of the communicating generated by the individual cumulative footprints. Example 9 in Section 3.6 discusses an instance of such a computation.

[2]For data partitioning, however, the formulation must be modified slightly. Data partitioning is the problem of partitioning the data arrays into data tiles and assigning the tiles to memory modules associated with the processing nodes so that a maximum number of the data references made by corresponding loop tiles are satisfied by the local memory module. Because data partitioning assumes that data from other memory modules is not dynamically copied locally (as in systems with caches), we replace the $max - min$ formulation by the *cumulative spread* a^+ of a set of uniformly intersecting references, whose k^{th} element is given by,

$$a_k^+ = \sum_r [a_{r,k} - med_r(a_{r,k})], \forall k \in 1, \ldots, d.$$

where $med_r(a_{r,k})$ is the median of the offsets in the k^{th} dimension. The rest of our framework applies to data partitioning if $\hat{a}$ is replaced by a^+. A more detailed description of data partitioning, however, is beyond the scope of this paper.

3.6 Minimizing the Size of the Cumulative Footprint

We now focus on the problem of finding the loop partition that minimizes the size of the cumulative footprint. Let us illustrate this problem through the following two examples.

Example 8

```
Doall (i, 1, N)
  Doall (j, 1, N)
    Doall (k, 1, N)
      A(i,j,k) = B(i-1,j,k+1) + B(i,j+1,k) + B(i+1,j-2,k-3)
    EndDoall
  EndDoall
EndDoall
```

Here we have two uniformly intersecting sets of references: one for A and one for B. Let us look at the class corresponding to B since it is more instructive. Because A has only one reference, its footprint size is independent of the loop partition, given a fixed total size of the loop tile, and therefore need not figure in the optimization process. The G matrix corresponding to the references to B is,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The $\hat{a}$ vector is $(2,3,4)$. Consider a rectangular partition $\mathbf{L} = \mathbf{\Lambda}$ given by

$$\begin{bmatrix} L_i & 0 & 0 \\ 0 & L_j & 0 \\ 0 & 0 & L_k \end{bmatrix}$$

In this example, the **LG** matrix is the same as the **L** matrix. The size of the cumulative footprint for B according to Theorem 2 is

$$L_iL_jL_k + 2L_jL_k + 3L_iL_k + 4L_iL_j$$

This expression must be minimized keeping $|\det \mathbf{L}|$ (or the product $L_iL_jL_k$) a constant. The product represents the area of the loop tile and must be kept constant to ensure a balanced load. The constant is simply the total area of the iteration space divided by P, the number of processors. For example, if the loop bounds are I, J, and K, then, we must minimize $L_iL_jL_k + 2L_jL_k + 3L_iL_k + 4L_iL_j$, subject to the constraint $L_iL_jL_k = IJK/P$.

This optimization problem can be solved using standard methods, for example, using the method of Lagrange multipliers [11]. The size of the cumulative footprint is minimized when L_i, L_j, and L_k are chosen in the proportions 2, 3, and 4, or

$$L_i : L_j : L_k :: 2 : 3 : 4$$

Abraham and Hudak's algorithm [6] gives an identical partition.

Thus far, our analysis has been concerned with minimizing the cumulative footprint size, which at first glance, appears to minimize the number of first-time cache misses. However, the

```
Doseq (t, 1, T)
  Doall (i, 1, N)
    Doall (j, 1, N)
      Doall (k, 1, N)
        A(i,j,k) = B(i-1,j,k+1) + B(i,j+1,k) + B(i+1,j-2,k-3)
      EndDoall
    EndDoall
  EndDoall
EndDoseq
```

Figure 9: A doall loop nest within a sequential loop.

same optimization process minimizes the number of coherence-related invalidations and misses as well. For example, consider the loop nest in Figure 9. In a load-balanced partitioning, $|\det \mathbf{L}|$ is a constant, so the $L_i L_j L_k$ term drops out, and the optimization process minimizes the volume of coherence traffic (given by $2L_jL_k + 3L_iL_k + 4L_iL_j$) for a given tile.

We now use an example to show how to minimize the total number of cache misses when there are multiple uniformly intersecting sets of references. The basic idea here is that the references from each set contributes additively to traffic.

Example 9

```
Doall (i, 1, N)
  Doall (j, 1, N)
    A(i,j) = B(i-2,j) + B(i,j-1) + C(i+j,j) + C(i+j+1,j+3)
  EndDoall
EndDoall
```

Let the tile $\mathbf{L}$ be

$$\begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}.$$

There are three uniformly intersecting classes of references, one for B, one for C, and one for A. As before, we can ignore A in the optimization process. The **LG** corresponding to references to array B is same as $\mathbf{L}$ and the size of the corresponding cumulative footprint is

$$\left|\det \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}\right| + \left|\det \begin{bmatrix} 2 & 1 \\ L_{21} & L_{22} \end{bmatrix}\right| + \left|\det \begin{bmatrix} L_{11} & L_{12} \\ 2 & 1 \end{bmatrix}\right|.$$

Similarly, **LG** for array C is

$$\begin{bmatrix} L_{11} + L_{12} & L_{12} \\ L_{21} + L_{22} & L_{22} \end{bmatrix},$$

and the size of the cumulative footprint with respect to C is

$$\left|\det\begin{bmatrix} L_{11}+L_{12} & L_{12} \\ L_{21}+L_{22} & L_{22} \end{bmatrix}\right| + \left|\det\begin{bmatrix} 1 & 3 \\ L_{21}+L_{22} & L_{22} \end{bmatrix}\right| + \left|\det\begin{bmatrix} L_{11}+L_{12} & L_{12} \\ 1 & 3 \end{bmatrix}\right|.$$

The problem of minimizing the size of the footprint reduces to finding the elements of $\mathbf{L}$ that minimizes the sum of the two expressions above subject to the constraint the $|\det \mathbf{L}|$ is a constant. This is a nonlinear optimization problem in general, and can be solved using numerical methods.

For the purpose of illustration, let us restrict the loop tiles to be rectangular. In other words we assume L_{12} and L_{21} to be zero. The total size of the cumulative footprint simplifies to $2L_{11}L_{22}+4L_{11}+6L_{22}$. The optimal values for L_{11} and L_{22} can be shown to satisfy the equation $4L_{11} = 6L_{22}$, using the method of Lagrange multipliers.

3.7 Rectangular Tiles

This section considers the special case when loop tiles are restricted to be rectangular. The special case of rectangular loop tiles is useful to consider for many reasons: (1) rectangular tiles lead to much more accurate estimates of communication volume, (2) they allow us to handle much more general forms of $\mathbf{G}$, and (3) they allow easy code generation.

We first prove a simple theorem on lattices and then state our result on the size of cumulative footprints for rectangular tiles.

Definition 9 *Given n linearly independent vectors $\vec{a}_1, \ldots, \vec{a}_n \in \mathcal{Z}^n$ the lattice $\mathcal{L}(\vec{a}_1, \ldots, \vec{a}_n)$ is*

$$\{\vec{x} = \sum_{i=1}^{n} l_i\vec{a}_i | l_i \in \mathcal{Z}, i = 1, \ldots, n\}.$$

We define $\mathcal{L}(\vec{a}_1, \ldots, \vec{a}_n, \lambda_1, \ldots, \lambda_n)$ to be

$$\{\vec{x} = \sum_{i=1}^{n} l_i\vec{a}_i | l_i \in \mathcal{Z}, 0 \leq l_i \leq \lambda_i\}.$$

We refer to $\mathcal{L}(\vec{a}_1, \ldots, \vec{a}_n, \lambda_1, \ldots, \lambda_n)$ as a bounded lattice.

We will reduce the problem of computing the size of the cumulative footprint to the problem of computing the size of the union of a bounded lattice and a translation of the lattice.

Theorem 3 *Let $L_1 = \mathcal{L}(\vec{a}_1, \ldots, \vec{a}_l, \lambda_1, \ldots, \lambda_l)$ and let L_2 be the translation of L_1 by vector $\vec{t}$. $L_1 \cap L_2$ is non-empty if and only if $\vec{t} = u_1\vec{a}_1 + u_2\vec{a}_2 + \ldots + u_l\vec{a}_l$ for some vector $\vec{u} \in \mathcal{Z}^l$ where $0 \leq u_i \leq \lambda_i$.*

Lemma 3 *Let $L_1 = \mathcal{L}(\vec{a}_1, \ldots, \vec{a}_l, \lambda_1, \ldots, \lambda_l)$ and let L_2 be the translation of L_1 by vector $\vec{t}$, $\vec{t} = u_1\vec{a}_1 + u_2\vec{a}_2 + \ldots + u_l\vec{a}_l$ for some vector $\vec{u} \in \mathcal{Z}^l$ where $0 \leq u_i \leq \lambda_i$.*

$$|L_1 \cup L_2| = 2\prod_{j=1}^{l}(\lambda_j + 1) - \prod_{j=1}^{l}(\lambda_j + 1 - u_j) \approx \prod_{j=1}^{l}(\lambda_j + 1) + \sum_{i=1}^{l} u_i \prod_{j=1,\ldots,l, j\neq i}(\lambda_j + 1) - \prod_{i=1}^{l} u_i$$

Theorem 4 *Let* **L** *be a rectangular partition and let the reference matrix* **G** *be nonsingular with rows* $\vec{g}_1, \vec{g}_2, \ldots, \vec{g}_l$. *The size of the cumulative footprint with respect to a set of uniformly intersecting references specified by the reference matrix* **G** *and a spread vector* $\hat{a}$ *is approximately*

$$\prod_{j=1}^{l}(L_{jj}+1) + \sum_{i=1}^{d} u_i \prod_{j=1,\ldots,l, j\neq i} (L_{jj}+1)$$

where u_i*s are given by* $\hat{a} = \sum_{i=1}^{l} u_i \vec{g}_i$.

It is to be noted that the condition on **G** is that it is just nonsingular and not necessarily unimodular. As before, this theorem can be applied by replacing **G** by a maximal set of independent columns of **G** when the columns of **G** are not independent.

The total communication is computed by adding the communication for each of the equivalence classes.

Example 10

```
Doall (i, 1, N)
  Doall (j, 1, N)
    A(i,j) = B(i+j,i-j) + B(i+j+4,i-j+2) + C(i,2i,i+2j-1) + C(i+1,2i+2,i+2j+1)+C(i,2i,i+2j+1)
  EndDoall
EndDoall
```

Let the rectangular tile **L** *be,*

$$\begin{bmatrix} L_i & 0 \\ 0 & L_j \end{bmatrix}.$$

We need to consider all the uniformly intersecting classes.

1. $B(i+j, i-j)$ *and* $B(i+j+4, i-j+2)$ *are uniformly intersecting. The G matrix corresponding to this class is,*

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Note that **G** *is nonsingular, but not unimodular. The* $\hat{a}$ *vector is* $(4,2) = 3(1,1)+1(1,-1)$. *The size of the cumulative footprint according to Theorem 4 is*

$$(L_i+1)(L_j+1) + 3(L_j+1) + (L_i+1)$$

2. $C(i, 2i, i+2j-1)$ *and* $C(i, 2i, i+2j+1)$ *are uniformly intersecting. Although* $C(i+1, 2i+2, i+2j+1)$ *belongs to the same uniformly generated set of references as* $C(i, 2i, i+2j-1)$ *and* $C(i, 2i, i+2j+1)$, $C(i+1, 2i+2, i+2j+1)$ *does not intersect with either one of the other two references to the array C. This is easily seen from Theorem 3. Here* **G** *is singular. However we can pick the first and third column of* **G** *and apply Theorem 4. The size of the cumulative footprint of the current class can be shown to be*

$$(L_i+1)(L_j+1) + (L_i+1).$$

3. *The two separate classes $C(i+1, 2i+2, i+2j+1)$ and $A(i,j)$ do not play any role in the optimal choice of L_i and L_j since the footprint for these classes equal the size of the tile which is constrained to be a constant for load balancing reasons.*

The problem of minimizing the size of the footprint reduces to finding L_i and L_j that minimizes

$$2(L_i + 1) + 3(L_j + 1)$$

subject to the constraint that $(L_i + 1)(L_j + 1)$ is a constant. The optimal values for L_i and L_j can be shown to satisfy the equation $2L_i = 3L_j + 1$, using the method of Lagrange multipliers.

3.8 General case of G

In this section we summarize our results relating to footprints and cumulative footprints when conditions on the reference matrix **G** are relaxed.

Theorem 5 *The size of the footprint of a tile* **L** *with respect to the reference* **G** *is equal to the number of lattice points in the tile* **L** *if the rows of* **G** *are linearly independent.*

The size of the footprint can be computed precisely for a general **G** in the following cases when the tile **L** is rectangular.

1. The loop nesting $l = 1$.
2. The loop nesting $l = 2$.
3. The loop nesting $l = 3$, the dimension of the array $d >= 2$ and the column rank of **G** is $>= 2$.

The above cases cover most of the practical situations. For the case when $l = 3$ and $d = 1$, it seems difficult to express the size of the footprint by a closed form expression. However, one can compute the exact size of the footprint efficiently using a table lookup when the elements of **G** are small, which is mostly the case in practice. We believe the same techniques can be extended for higher dimensions as well. We have similar results for computing the cumulative footprint as well.

4 Implementation

We have implemented some of the ideas from our framework in a compiler for the Alewife machine [12]. The Alewife machine implements a shared global address space with distributed physical memory and coherent caches. The nodes are configured in a 2-dimensional mesh communication network. Distributed-memory architectures may require three types of related analyses to distribute code and data on to the machine:

Loop Partitioning Each processor must be assigned a set of loop iterations that maximizes reuse of data in caches and achieves good load balance.

Data Partitioning and Alignment Arrays must be distributed among the processors such that memory references that miss in the cache go to the local memory rather than accross the network to another node. This is accomplished by partitioning arrays with the same aspect ratios as the iterations of loops that reference them, and then assigning corresponding loop and data partitions to the same processor.

Placement In an architecture like Alewife the memory access time depends on the distance between the node making the memory request and the node where the requested data resides. The data partitioning and alignment phases make assignments to virtual processors which must be mapped onto the real machine in order to minimize memory reference latency. This is a smaller effect that may become important in very large machines.

We have implemented loop and data partitioning as well as alignment for the case of rectangular partitioning. The structure of our compiler is shown in Figure 10. The input to the compiler is a program where paralellism is specified either by the programmer, or in a previous compilation phase. As in [6], we separate the notion of parallelization from that of implementation. The languages accepted at present are Mul-T, a parallel Lisp language, and Semi-C, a parallel version of C. An initial series of transformations are performed including constant-folding and procedure integration producing a graphical intermediate form called WAIF.

WAIF is a hierarchical graphical representation of a source program. Functionally as well as structurally, WAIF has two abstraction levels: The program graph with dependence edges (WAIF-PG) and the abstracted task and data communications graph (WAIF-CG). WAIF-PG is a customized version of an abstract syntax tree. WAIF-CG summarizes the communication patterns between tasks and data structures that can be derived from a static analysis. Data and loop partitioning are performed as transformations on the WAIF-CG and then code for sequential threads with explicit synchronization is generated. The sequential code-generation process performs standard optimizations such as strength reduction and loop-invariant code motion, producing machine code for Alewife's processors. the Alewife machine.

Given the presented framework, one can take various examples and calculate the reduction in expected cache misses when better loop partitions are used. The question is what the impact is on overall performance. In our case, the effects of distributed memory (data partitioning and alignment) are important and we were unable to isolate the effect of cache miss reduction in time for this version of the paper. Making this measurement is complicated because changing the loop partition alters the number of non-local memory references as well as affecting cache behavior. We plan to have some results for the final version of the paper.

5 Related Work

We show that Abraham and Hudak's results on rectangular loop partitioning for caches [6] (where the index expressions are of the form index variable plus a constant) are reproduced by our theoretical framework (see Example 8). The framework, however, is able to handle much more general index expressions, and produce parallelogram partitions if desired.

We also show that the framework correctly produces the communication-free loop partitions for the class of programs handled by Ramanujam and Sadayappan [7]. In addition, the same framework is able to discover optimal partitions in cases where communication free partitions are not possible – a case not handled by [7]. (See Examples 2 and 10.)

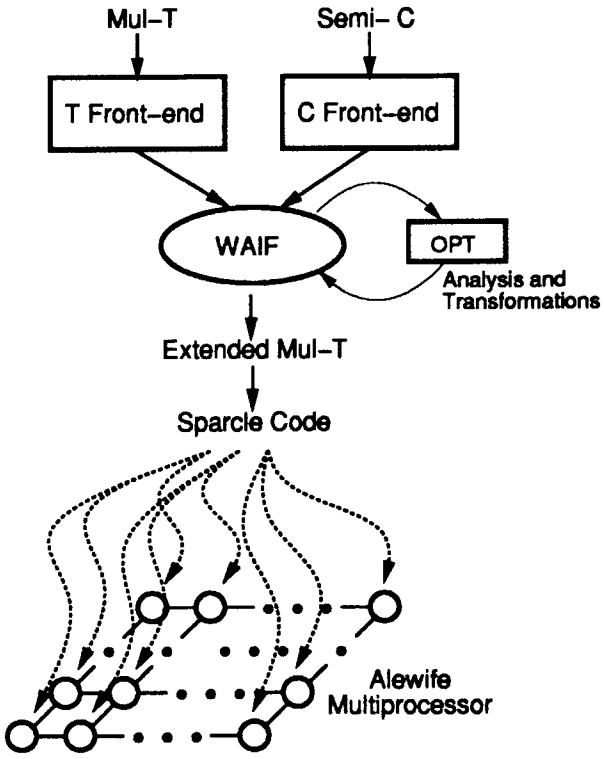

Figure 10: The Alewife Code Generation Process.

Ferrante, Sarkar, and Thrash [5] address the problem of estimating the number of cache misses for a nest of loops. This problem is similar to our problem of finding the size of the cumulative footprint, but differs in these ways: (1) We consider a tile in the iteration space and not the entire iteration space; our tiles can be hyperparallelepipeds in general. (2) We partition the references into uniformly intersecting sets, which makes the problem computationally more tractable, since it allows us to deal with only the tile at the origin. (3) Our treatment of coupled subscripts is much simpler, since we look at maximal independent columns, as shown in Section 3.4.1. (4) Finally, our techniques yield better estimates for references of the form $A[i+j+k, 2i+3j+4k]$.

Our work complements the work of Wolfe and Lam [4]. They use the notion of equivalence classes within the set of uniformly generated references to identify valid loop transformations to maximize the degree of temporal and spatial locality within a given loop nest, but they do not attempt to derive optimal iteration space partitions, as we do.

6 Acknowledgements

This research is supported by Motorola Cambridge Research Center and by NSF grant # MIP-9012773. Partial support has also been provided by DARPA contract # N00014-87-K-0825, in part by a NSF Presidential Young Investigator Award.

7 Conclusions

The performance of cache-coherent systems is heavily predicated on the degree of temporal locality in the access patterns of the processor. If each block of data is accessed a number of times by a given processor, then caches will be effective in reducing network traffic. Loop partitioning for cache-coherent multiprocessors strives to achieve precisely this goal.

This paper presented a theoretical framework to derive the aspect ratios of the iteration-space partitions of the do loops to minimize the communication traffic in multiprocessors. The framework allows the partitioning of doall loops into hyperparallelepiped tiles where the index expressions in array accesses can be any affine function of the indices.

Our analysis introduces the notion of uniformly intersecting references to categorize the references within a loop into classes that will yield cache locality. The notion of data footprints is introduced to capture the combined set of data references made by the reference within each uniformly intersecting class. Then, an algorithm to compute the total size of the data footprint for a given loop partition is presented. Once an expression for the total size of the data footprint is obtained, standard optimization techniques can be applied to minimize the size of the data footprint and derive the optimal loop partitions.

The framework correctly reproduces results from loop partitioning algorithms previously proposed by other researchers. In addition, it discovers optimal partitions in many more general cases that are not handled by previous algorithms.

A subset of the framework, including both loop and data partitioning has been implemented in the compiler system for the Alewife multiprocessor. The implementation currently handles rectangular partitions only. Integrating the rest of the framework into our compiler system is part of an ongoing effort.

References

[1] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[2] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[3] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[4] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference Programming Language Design and Implementation*, pages 30–44, 1991.

[5] J. Ferrante, V. Sarkar, and W. Thrash. *On Estimating and Enhancing Cache Effectiveness*, pages 328–341. Springer-Verlag, August 1991. Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing. Editors U. Banerjee and D. Gelernter and A. Nicolau and D. Padua.

[6] S. G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[7] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

```
Doall (i, 1, N)
  Doall (j, 1, N)
    Doall (k, 1, N)
      l$C[i,j] = l$C[i,j] + A[i,k] + B[k,j]
    EndDoall
  EndDoall
EndDoall
```

Figure 11: Structure of a single loop nest

[8] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of Supercomputing '91*. IEEE Computer Society Press, 1991.

[9] G. N. Srinivasa Prasanna, Anant Agarwal, and Bruce R. Musicus. Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory. To appear in IEEE Transactions on Parallel and Distributed Systems. Also available as MIT Laboratory for Computer Science TM-466, June 1992.

[10] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1990.

[11] George Arfken. *Mathematical Methods for Physics*. Academic Press, 1985.

[12] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[13] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, August 1991.

[14] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

A Handling Synchronization References

Fine-grain data-level synchronization can sometimes be represented within a parallel do loop and its cost approximately modeled as slightly more expensive communication than usual [9]. For example, in the Alewife system the inner loop of matrix multiply can be written using fine-grain synchronization in the form of the loop in Figure 11.

In the code segment in Figure 11, the "l$" preceding the **C** matrix references denote atomic accumulates. Accumulates into the **C** array can happen in any order, just that each accumulate action must be atomic. Such synchronizing reads or writes are both treated as writes by the coherence system. Similar linguistic constructs are also present in Id [13] and in a variant of FORTRAN used on the HEP [14].

B Examples of Uniformly Intersecting References

Example 11 *The following sets of references are uniformly intersecting.*

1. $A[i, j], A[i + 1, j - 3], A[i, j + 4]$.
2. $A[2j, 2, 4], A[2j - 5, 2, 3], A[2j + 3, 2, 1]$.

The following pairs are not uniformly intersecting.

1. $A[i, j], A[2i, j]$.
2. $A[i, j], A[2i, 2j]$.
3. $A[j, 2, i], A[j, 3, i]$.
4. $A[2i], A[2i + 1]$.
5. $A[i + 2, 2i + 4], A[i + 5, 2i + 6]$.
6. $A[i, j], B[i, j]$.