Estimating Cache Misses and Locality Using Stack Distances

Calin Caşcaval
IBM TJ Watson Research Center
Yorktown Heights, NY
cascaval@us.ibm.com

David A. Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
padua@cs.uiuc.edu

ABSTRACT

Cache behavior modeling is an important part of modern optimizing compilers. In this paper we present a method to estimate the number of cache misses, at compile time, using a machine independent model based on stack algorithms. Our algorithm computes the stack histograms symbolically, using data dependence distance vectors and is totally accurate when dependence distances are uniformly generated. The stack histogram models accurately fully associative caches with LRU replacement policy, and provides a very good approximation for set-associative caches and programs with non-constant dependence distances.

The stack histogram is an accurate, machine-independent metric of locality. Compilers using this metric can evaluate optimizations with respect to memory behavior. We illustrate this use of the stack histogram by comparing three locality enhancing transformations: tiling, data shackling and the product-space transformation. Additionally, the stack histogram model can be used to compute optimal parameters for data locality transformations, such as the tile size for loop tiling.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers,Optimization*; C.4 [**Computer System Organization**]: Performance of Systems—*Modeling Techniques*

General Terms

Algorithms, Performance, Measurement

Keywords

stack algorithms, compiler algorithms, cache modeling

1. INTRODUCTION

The *Memory Wall* problem of current architectures has been well documented. Processor speeds increase much faster than memory speed, applications access larger and larger data sets. As a result, programming paradigms are shifting. A number of hardware and software techniques have been developed to reduce the increasing impact of memory latency. Examples of hardware techniques are caching (instructions and data), out-of-order execution, multithreaded execution, prefetching, processors in memory (PIM) and speculation. Of these techniques, by far the most common is caching. On the software side, there is a plethora of compiler optimizations targeted toward hiding or eliminating memory latencies. Insertion of prefetch instructions [22], instruction scheduling and reordering [12], and loop parallelization for SMT [17] are some of the latency hiding transformations. Loop transformations that enhance data locality, such as tiling [7, 33, 34, 16], and fusion [18], or code transformations such as data shackling [14] and the product-space transformation [1] are techniques to increase data reuse.

These compiler transformations rely, sometimes implicitely, on models of memory behavior to predict performance in order to select which optimizations are applied and the parameters for the transformations. In this paper, we introduce a method to model the memory behavior of programs and present three applications of the method: evaluation of compiler transformations, automatic tile size selection, and cache miss estimation.

Our method is capable of handling with total accuracy 80% of the loops in the SPECfp95 benchmarks. When compared with two of the most recent published methods of statically estimating cache misses (Ghosh et al. [11] and Chatterjee et al. [6]) our method stands out as practical. For example, Chatterjee's model, while theoretically able to handle a larger class of codes than ours, in practice is limited by its running time. One advantage over [11] is that our method is machine independent - we compute the stack histogram once and then use it to compute the cache misses for several cache sizes. This property facilitates the use of our method for static and dynamic decisions.

The rest of the paper is organized as follows: in Section 2 we present our compile-time algorithm to compute stack histograms. Section 3 shows how the stack histogram can be used to estimate the effectiveness of data locality enhancing optimizations. In the same section we compare the number of cache misses predicted at compile-time using our technique against cache misses measurements taken using the hardware counters on the MIPS R10000 processor for a subset of the SPECfp95 benchmarks. We present related work in Section 4 and conclude in Section 5.

ICS'03 June 23–26, 2003, San Francisco, California, USA

2. COMPILE-TIME STACK HISTOGRAM COMPUTATION

The memory model is based on the stack histogram obtained from the stack processing algorithm [19], which consumes a trace of memory references in a program, and builds a stack and its histogram $S(\Delta)$ as follows: (1) if the reference is the first access to that trace element, set $\Delta = \infty$, increment $S(\infty)$ and push the reference on the stack; (2) if a trace element has been previously referenced, the reference will be already on the stack, let Δ be the distance from the top of the stack to the position at which the reference is found, increment $S(\Delta)$, and move the reference to the top of the stack, all the references between top and Δ being pushed down one position. The references below Δ are not affected. The references can be to memory locations, cache lines or memory pages.

The result of the stack processing algorithm is the histogram $S(\Delta)$ which is used to calculate the number of cache misses, for any cache size, as follows:

$$H(C) = \sum_{\Delta=0}^{C} S(\Delta) \quad \text{and} \quad M(C) = \sum_{\Delta=C+1}^{\infty} S(\Delta) \qquad (1)$$

Consider a physical memory (or cache) of size C. The sum of references at stack distances less than C, $H(C)$ in Eq. (1), is the number of in-core references (cache hits), whereas the sum of references at stack distances greater than C, $M(C)$ in Eq. (1), represents out-of-core accesses (cache misses).

Computing the stack histogram at run-time is time consuming. Even with very efficient algorithms [2], to obtain the histogram for a program, one has to run that program, therefore the machine has to be available. However, the stack histogram can often be computed at compile-time using data dependence information. Considering that compile-time information is restricted (loop bounds may be unknown, array memory layouts are approximated, etc.), we make several assumptions, which are discussed next.

The compiler needs to determine the memory trace generated by a loop in order to compute the stack histogram of that loop. This implies that the value of all subscript expressions are known at compile-time (symbolically), which would be impossible in the presence of certain classes of subscript expressions, such as subscripted subscripts (i.e., subscripts involving array elements). For a compiler to determine the actual memory locations accessed by array references involving subscripted subscripts would require the compiler to know the values of the arrays used in the subscript expressions. And, although the compiler can usually propagate the value of arrays when they are computed in the program [15], array element values are often functions of the input data and therefore cannot be statically known. Furthermore, most dependence tests assume affine subscript expressions. Therefore, our method is restricted to affine functions of the loop indices. The majority of the subscripts are of this type, for example, more than 80% of subscripts in the Perfect Club benchmarks and SPECfp95 benchmarks are affine [25]. Another common form is array elements as subscript expressions, and occurs mostly in sparse matrix operations. Methods that approximate cache misses for these types of indices have presented in [4].

To summarize, in order to obtain totally accurate stack histograms, we assume that the analyzed loops satisfy the following conditions: (i) all subscript expressions are affine expressions of the loop indices, (ii) there are no conditional instructions in the body of the loop, (iii) the loop limits are constants or loop invariants, and (iv) all dependences between array references are uniform [10]. Many of the compile-time cache models in the literature are similarly restricted. Our method can be used to model programs that do not satisfy all of these conditions with some loss of accuracy – the memory trace will be conservatively computed, thus not guaranteed to be exact [5].

The stack histogram can be used to predict the number of cache misses at different granularities in a program, i.e. loop nests, routines, or entire programs. However, the compile-time algorithm presented in this work focuses on generating stack histograms for separate loop nests. We consider that each loop nest starts with a cold cache, i.e., none of the array elements accessed in the loop are present in the cache at the first iteration. Further research needs to be done to enable the estimation for sequences of loop nests. The current implementation of the algorithm is limited by the power of the symbolic manipulator used to compute dependence spans and referenced sets.

The compile-time stack algorithm can estimate the number of misses for fully associative caches with the LRU replacement policy. In Section 3.2 we show how the number of misses for an LRU set-associative cache can be approximated from the LRU fully associative histogram using statistical methods. We obtained very good estimates (within 10%) when compared to actual measured misses for two way set-associative caches [4]. However, for programs with larger data set sizes than the benchmarks we studied, the accuracy could be significantly lower.

2.1 Notation

Before describing our algorithm we present a few definitions. Each occurrence of an array element in any expression in the body of a loop will be called an *array reference*. For example, the sequence of statements:

```
A(I)=B(I)+A(I-1)
C(I)=A(I)+C(K(I))
```

contains seven array references: $A(I-1)$, $B(I)$, $A(I)$, $K(I)$, $C(K(I))$, $A(I)$, and $C(I)$. Each dynamic execution of an array reference is called an instance of the array reference. Within loops, instances are identified by the values of the loop indices. For example, in the loop:

```
DO I=1,N
   DO J=1,M
      X(I,J)=X(I-1,J)+1
      D(I)=D(I)+X(I,J)
```

there are $N \times M$ instances of each one of the five array references, one for every possible value of the two loop indices. We will denote array references using italic lower case letters, and instances by italic letters extended with subscripts containing the value of loop indices. For example, the array reference $D(I)$ that occurs on the left hand side of the second statement is referred to as d. The first instance of d will be named $d_{\langle 1,1 \rangle}$ and the last instance $d_{\langle N,M \rangle}$.

Given two, not necessary distinct array references a and b surrounded by m loops, we say there is a *loop-carried dependence* from a to b if there are two instances $a_{\langle i_1,i_2,\ldots,i_m \rangle}$ and $b_{\langle j_1,j_2,\ldots,j_m \rangle}$, with iteration $\vec{i} = \langle i_1, i_2, \ldots, i_m \rangle$ executing before iteration $\vec{j} = \langle j_1, j_2, \ldots, j_m \rangle$, that refer to the same array element. We say that reference b depends on a and

also that instance $b_{\vec{j}} = b_{\langle j_1, j_2, \ldots, j_m \rangle}$ depends on instance $a_{\vec{i}} = a_{\langle i_1, i_2, \ldots, i_m \rangle}$. Also, we say that instance $b_{\vec{j}}$ is the *target* of the dependence and that $a_{\vec{i}}$ is its *source*. We call the vector $\vec{d} = \vec{j} - \vec{i} = \langle j_1 - i_1, j_2 - i_2, \ldots, j_m - i_m \rangle$, a *distance vector* of the dependence from a to b. We say that a cross-iteration dependence from reference a to reference b is uniform if given a distance vector of the dependence, $D = \vec{d}$, and any two iteration points $\vec{p} = \langle p_1, p_2, \ldots, p_m \rangle$ and $\vec{q} = \langle q_1, q_2, \ldots, q_m \rangle$ with $\vec{d} = \vec{q} - \vec{p}$, the instance $b_{\vec{q}}$ depends on the instance $a_{\vec{p}}$ and the distance vector $\vec{d}$ is constant (i.e., each element of $\vec{d}$ is invariant with respect to the loop indices). In this paper we use the term dependence as a placeholder for all types of dependences: input, flow, anti, and output.

2.2 Stack Histogram Computation

To generate the stack histogram at compile-time, we need to compute the stack distance for each reference and then accumulate these to compute the number of references that occur for each stack distance. To simplify the discussion, we will first assume that our memory trace will contain memory locations rather than cache lines or pages. This restriction will be removed later in Section 2.5.

Our compiler algorithm proceeds by computing the number of distinct array elements accessed from the source to the target of each dependence between array reference instances. If an array reference instance is the target of only one dependence, the stack distance will clearly be the number of distinct references accessed from the source to the target of the dependence. The reason is that the dependence will relate two consecutive references to the same location. If the array element instance is the target of more than one dependences the stack distance will be the minimum number of references associated with all the incoming data dependences. The reason is that the minimum dependence distance will correspond to consecutive accesses to the same memory location. The number of accesses at a stack distance is computed by counting the number of array reference instances that generate that particular stack distance.

2.3 An Example

Before describing the algorithm formally, we present a simple example to illustrate how the algorithm works. Consider the code fragment in Figure 1(a). In this loop there is one loop carried dependence, δ, from $A(I)$ to $A(I-2)$, with distance vector $D = \langle 2 \rangle$. As just mentioned, we need to compute $AS(\delta)$, the number of distinct array references between the source and the target of this dependence at iteration $\vec{k}$. The instance at iteration k with $5 \leq k \leq N$, of the $A(I-2)$ array reference will be the target of this dependence. And each one of these instances will generate a reference at a stack distance equal to $AS(\delta)$. The other two instances (those iterations 1 and 2) of reference to $A(I-2)$, which do not have an incoming dependence will generate references at distance ∞. The names a, a', and b, will be used to represent array references $A(I-2)$, $A(I)$, and $B(I)$, respectively.

After identifying the dependence, our algorithm proceeds as follows:

Step 1: *Partition the iteration space.* In our example the iteration space is partitioned into two elementary components. In the first elementary component (iterations 1 and 2), the instances of a have no incoming dependences, while in the second elementary component (iterations 3 to $N-2$) the dependence is present.

Step 2 : *Compute $AS(\delta)$ for each dependence in each elementary component.* In our example there is only one dependence and we only need to consider the second elementary component because there are no incoming dependences into the first elementary component.

Step 2.1: *Determine the dependence span, $DS(\delta, k)$, the set of iterations spanned by the dependence δ in iteration k.* In this example, $DS(\delta, k) = \{k-2, k-1, k\}$ for array reference instance a_k, with $k = 5, 6, \ldots, N$. Note that the dependence span is computed symbolically and it represents, in a single formula, all instances of the array reference targeted by the dependence.

Step 2.2: *Identify the array elements accessed in the $DS(\delta, k)$ iterations.* The algorithm computes the referenced set AR which, by definition, is the set of array elements accessed by all iterations in a dependence span. AR is therefore a function of the dependence and of the array reference instance. As shown in Figure 1(b), the first array reference $A(I-2)$ accesses memory locations $AR(a_k, \delta) = \{A(k-2), A(k-3)\}$, $A(I)$ accesses locations $AR(a'_k, \delta) = \{A(k-2), A(k-1)\}$, and the locations accessed by $B(I)$ are $AR(b_k, \delta) = \{B(k-1)\}$. Again, the AR sets are computed symbolically to represent, in a single formula, the memory locations accessed by all instances of an array reference. The number of array elements accessed during all iterations is the cardinality of the union of all ARs. In our case, $AS(\delta) = |AR(a_k, \delta) \cup AR(a'_k, \delta) \cup AR(b_k, \delta)| = |\{A(k-1), A(k-2), A(k-3), B(k-1)\}| = 4$. Notice that although the ARs are function of the iteration k, AS does not depend on the iteration because it is constant for all iterations of each elementary component of the iteration space.

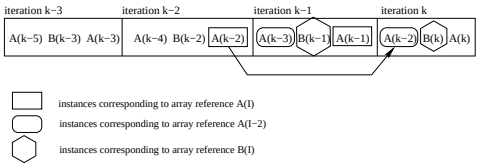
Step 3: *Compute the histogram.* For each array reference we consider the incoming dependences in each elementary component of the iteration space. Thus, $A(I)$ does not have any incoming dependence in any partition, therefore it contributes $N-2$ to ∞, because it is the first reference to that array element in every iteration. $B(I)$ contributes $N-2$ to ∞. The reference $A(I-2)$ is not the target of any dependences in the first elementary component, whose size is 2 iterations, therefore it contributes 2 to ∞. It has δ as incoming dependence in the second elementary component, thus, it contributes $N-4$ to distance 4, because there are four accesses to different array elements between the dependent accesses.

2.4 Algorithm Description

In this section we describe the compile-time stack histogram computation algorithm formally. The expressions used are all symbolic, thus the following algorithms rely on the compiler's implementation of computer algebra operations.

```
do i = 3, N
   A(i) = A(i-2) + B(i)
enddo
```

(a) Fortran Code



(b) Iteration space, array references and instances

$$\begin{array}{ll} 4 & \leftarrow N-4 \\ \infty & \leftarrow 2N-2 \end{array}$$

(c) Stack histogram computation

Figure 1: Symbolic stack histogram example

2.4.1 Iteration Space Partitioning

The first step of the algorithm is to partition the iteration space in such way that, within each partition, all instances of each array reference have exactly the same incoming dependences. This is accomplished by separating iterations at either the beginning or the end of the set of iterations of some or all the loops in a nest. The example above illustrates such a partitioning. Based on the distance vector $D(\delta) = \langle 2 \rangle$ it was determined that δ does not reach the array reference instances accessed in the first 2 iterations of the loop. In general, consider the iteration space of a loop nest consisting of m loops:

```
DO I1 = L1, U1
   DO I2 = L2, U2
      ...
         DO Im = Lm, Um
         END DO
   END DO
END DO
```

Without loss of generality, and to simplify the discussion, we will assume that the step of all loop indices is 1. This iteration space is a set of tuples of the form $\langle i_1, i_2, \ldots, i_m \rangle$, where i_k is the index value of the kth nested loop. We represent this set as:

$$R = \bigodot_{k=1}^{m} (I_k \leftarrow [L_k, U_k]) = (I_1 \leftarrow [L_1, U_1]) \odot (I_2 \leftarrow [L_2, U_2]) \odot \cdots \odot (I_m \leftarrow [L_m, U_m]),$$

where L_k and U_k are constants or affine expressions of constants and loop indices, I_j, with $j < k$, and the operator $\odot$ represents the cross product of all iterations. For example, $(i \leftarrow [1,2]) \odot (j \leftarrow [1,2])$ represents the set $\{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle, \langle 2,2\rangle\}$. When $L_k = U_k = i_k$ we denote the term as $(I_k \leftarrow [i_k])$. If any of the loops have $U_k < L_k$, we assume that the iteration space expression for that loop is the empty set.

A distance vector $D(\delta) = \langle d_1, d_2, \ldots, d_m \rangle$, from array reference a to array reference b, partitions the iteration space R into two subsets. One subset is iteration space represented by

$$Q = \bigodot_{k=1}^{m} \left(I_k \leftarrow \left[L_k + d_k^+, U_k - d_k^-\right]\right),$$

$$\text{where } d_k^+ = \begin{cases} d_k & \text{if } d_k > 0 \\ 0 & \text{otherwise} \end{cases},$$

$$\text{and } d_k^- = \begin{cases} |d_k| & \text{if } d_k < 0 \\ 0 & \text{otherwise} \end{cases}.$$

In other words, Q is computed by "peeling off" for all $k \leq m$ the first d_k iterations or the last d_k iterations of the kth loop, depending on the sign of d_k. By peeling off these iterations we can guarantee that all instances $a_{\langle i_1, i_2, \ldots, i_m \rangle}$, such that $\langle i_1, i_2, \ldots, i_m \rangle \in Q$, will be actual targets of the dependence δ. The other subset, RMQ, will generate instances $a_{\langle i_1, i_2, \ldots, i_m \rangle}$ that will not be target of the dependence. It has the following form:

$$RMQ = R - Q = \bigcup_{x=1}^{m} \left(\bigodot_{k=1}^{x-1} \left(I_k \leftarrow \left[L_k + d_k^+, U_k - d_k^-\right]\right) \right) \odot P \odot \left(\bigodot_{k=x+1}^{m} (I_k \leftarrow [L_k, U_k]) \right),$$

$$\text{where } P = \begin{cases} (I_x \leftarrow [L_x, L_x + d_x]) & \text{if } d_x > 0 \\ (I_x \leftarrow [U_x - |d_x|, U_x]) & \text{otherwise} \end{cases}.$$

For $x = 1$ the term $\bigodot_{k=1}^{x-1} \left(I_k \leftarrow \left[L_k + d_k^+, U_k - d_k^-\right]\right)$ will be absent from the expression and for $x = m$ the term $\bigodot_{k=x+1}^{m} (I_k \leftarrow [L_k, U_k])$ will be absent. The partitioning algorithm, presented in Figure 2, operates on a set of iteration spaces. The set, $PartIS$, initially contains the whole iteration space. The algorithm proceeds by iterating over all distance vectors, and use the sets Q and RMQ generated by the the distance vector to refine the set $PartIS$ by partitioning each component of $PartIS$ into two subsets.

All components of set $PartIS$ will be of the form $\bigodot_{k=1}^{m} \left(I_k \leftarrow \left[L_k + d_k^1, U_k - d_k^2\right]\right)$ with all d_k^1s and d_k^2s constants. The intersection can therefore be computed as:

$$\bigodot_{k=1}^{m} \left(I_k \leftarrow \left[L_k + d_k^1, U_k - d_k^2\right]\right) \cap \bigodot_{k=1}^{m} \left(I_k \leftarrow \left[L_k + d_k^3, U_k - d_k^4\right]\right)$$

$$= \bigodot_{k=1}^{m} \left(I_k \leftarrow \left[L_k + max(d_k^1, d_k^3), U_k - min(d_k^2, d_k^4)\right]\right)$$

The number of partitions is in general $O(2^N)$ where N is the number of loop-carried dependences in the loop, but in practice it is usually much smaller because dependences often generate the same partitions.

2.4.2 Dependence Spans

Next, we need to compute the dependence spans which are the set of iteration points between the source iteration

$$PartIS = \left\{ \bigodot_{k=1}^{m} (I_k \leftarrow [L_k, U_k]) \right\}$$

foreach direction vector $D(\delta) = \langle d_1, d_2, \ldots, d_m \rangle$

let x be the index of the loop carrying the dependence

$$P = \begin{cases} (I_x \leftarrow [L_x, L_x + d_x]) & \text{if } d_x > 0 \\ (I_x \leftarrow [U_x - |d_x|, U_x]) & \text{otherwise} \end{cases}$$

$$R = \bigodot_{k=1}^{m} \left(I_k \leftarrow \left[L_k + d_k^+, U_k - d_k^-\right]\right)$$

$$RMQ = \bigcup_{x=1}^{m} \left(\bigodot_{k=1}^{x-1} \left(I_k \leftarrow \left[L_k + d_k^+, U_k - d_k^-\right]\right) \right) \odot P \odot \left(\bigodot_{k=x+1}^{m} (I_k \leftarrow [L_k, U_k]) \right)$$

foreach $s \in PartIS$

$PartIS = PartIS - s$

$PartIS = PartIS \cup \{s \cap R\} \cup \{s \cap RMQ\}$

end foreach

end foreach

Figure 2: Iteration space partitioning algorithm

and the target iteration of each dependence distance vector. In other words, the dependence span of a distance vector $D(\delta) = \vec{d} = \langle d_1, d_2, \ldots, d_m \rangle$ is the set of iterations that start at $\vec{i} = \langle i_1, i_2, \ldots, i_m \rangle$ and end at $\vec{i} + \vec{d} = \langle i_1 + d_1, i_2 + d_2, \ldots, i_m + d_n \rangle$, as long as both are in the iteration space. Geometrically, the dependence span can be conceived as a hull in the iteration space that encloses all these iteration points. Under the assumption that $d_1 > 0$ it is easy to see that the dependence span is the union of three sets, T, U, and V, such that $DS(\delta, \vec{i} + \vec{d}) = T \cup U \cup V$. The first set, T, represents all "beginning" iterations, when one or more of the outermost loop indices are set at their initial value:

$$T = \bigcup_{x=2}^{m+1} \left(\bigodot_{k=1}^{x-1} (I_k \leftarrow [i_k]) \right) \odot (I_x \leftarrow [i_x + 1, U_x]) \odot \left(\bigodot_{k=x+1}^{m} (I_k \leftarrow [L_k, U_k]) \right)$$

The term $\left(\bigodot_{k=x+1}^{m} (I_k \leftarrow [L_k, U_k])\right)$ will be absent when $x \geq m$ and the term $(I_x \leftarrow [i_x + 1, U_x])$ will be absent in the previous expression when $x = m$.

The second set, U, is the middle part of the iteration space:

$$U = (I_1 \leftarrow [i_1 + 1, i_1 + d_1 - 1]) \odot \left(\bigodot_{k=2}^{m} (I_k \leftarrow [L_k, U_k]) \right)$$

and the third set, V, is the tail, where at least one of the outermost loops has reached its upper limit.

$$V = \bigcup_{x=2}^{m+1} \left(\bigodot_{k=1}^{x-1} (I_k \leftarrow [i_k + d_k]) \right) \odot (I_x \leftarrow [L_x, i_x + d_x - 1]) \odot \left(\bigodot_{k=x+1}^{m} (I_k \leftarrow [L_k, U_k]) \right).$$

2.4.3 Referenced Sets Computation

A referenced set $AR(a_{\vec{k}}, \delta)$ (defined in Section 2.3) is computed by substituting in the subscript expression of a the ranges of the iteration variables taken from the dependence span. In geometrical terms, a referenced set is the projection of the dependence span on the array space using the subscript expression. For example, the array reference $A(2I + 1, J)$ for the iteration space $(I \leftarrow [1, 20]) \bigodot (J \leftarrow [2, 10])$ corresponds to the set of elements represented by `A(3:21,2:10)`. Identifying the set of arrays accessed by a reference within a simple iteration space (i.e., a rectangular iteration space involving no union operations) when the subscript expressions involve a single loop index is straightforward. It is harder to compute an accurate result when several loop indices appear in the same subscript expression because interval arithmetic becomes non-trivial. The problem of mapping a static array reference to a set of array elements within an iteration space is outside the scope of this paper. We have used the interval arithmetic described in [5]. The reader is referred to [23] and [24] for a detailed discussion on this topic.

Whenever there are inaccuracies in computing the referenced sets, we mark the sets accordingly. The measure of accuracy propagates further in the cost model, such that, when the prediction expressions are evaluated, together with the performance estimation we provide a "confidence" measure for the prediction.

If a dependence spans more than one static reference to the same array, the referenced set contributed by that array is computed by taking the union over the referenced sets of individual references. The number of distinct array elements spanned by the dependence is the sum of the cardinalities of the union of the referenced sets for all arrays spanned by the dependence:

$$AS(\delta) = \sum_{A} \left| \bigcup_{r \in A} AR(r, \delta) \right|, \text{ for all arrays } A.$$

2.4.4 Stack Histogram

Once the referenced sets are computed for each dependence span in each partition, all data required to compute the stack histogram are available. The stack histogram is composed of two sets of values, the stack distances and the number of accesses at that particular stack distance.

Each array reference contribution to a stack distance is either determined by its incoming dependences, or ∞ if there are no incoming dependences (the reference corresponds to a cold miss). The number of accesses contributed by each

foreach partition $p \in PartIS$
  foreach static array reference r in the loop body

$$\text{let } \Delta = \begin{cases} min_\delta(AS(\delta)) & \text{if } \exists \delta \text{ such that} \\ & target(\delta) = r \text{ and} \\ & \delta \text{ is valid in } p \\ \infty & \text{otherwise,} \end{cases}$$

    $S(\Delta) += |p|$
  end foreach
end foreach

Figure 3: Stack histogram computation algorithm

array reference is determined by the number of dynamic executions of the reference. The algorithm to compute the stack histogram is shown in Figure 3. It essentially walks all the array references in the loop nest and finds the minimum AS on all the incoming dependences.

Once the stack histogram is computed symbolically it can be used for different applications as shown in Section 3.

2.5 Spatial Locality

In the previous discussion we considered the cache lines to be of only one array element. In order to compute the stack histogram for real cache line sizes, we need to determine the number of distinct cache lines that are spanned by a dependence. Since we already computed the number of distinct array elements spanned by a dependence (the references set AS), we just have to translate that number into cache lines. In other words, we need to determine the cache lines layout for the referenced sets. When information such as stride distances or data alignment is not statically known, the resulting stack distance must be approximated. Because the mapping of array elements to cache lines is not known at compile time, the resulting stack histogram is an approximation.

As an example, consider a two dimensional array A, with $MMAX \times NMAX$ elements, and a dependence that spans a $M \times N$ referenced set. Also assume that the array is mapped in column major order, as in Fortran.

We compute $LDA = \left\lceil \frac{MMAX}{LS} \right\rceil$, the number of cache lines that cover one column of the matrix, where LS is the size of the cache line expressed in number of array elements. The number of cache lines covering a referenced set with dimensions M and N is given by the equation:

$$DL = \sum_{i=1}^{N} \left\lceil \frac{(i \times LDA)\%LS + M + offset}{LS} \right\rceil \qquad (2)$$

where $offset = 0$ if the first element of the array maps at the beginning of a cache line.

The stack histogram is computed using the same algorithm presented in Section 2.4.4, except that the number of distinct cache lines accessed is computed. In other words, the referenced set size AS is returned in terms of cache lines. The expressions denoting both stack distances and array references contain symbolic variables to denote the cache line size. These symbolic variables are treated like all the other hardware parameters used in the performance expressions.

2.6 Associativity

It has been previously shown [28, 13] that set-associative miss ratios can be estimated from the fully-associative miss-ratio. The compile-time stack distances algorithm estimates the number of misses for fully-associative LRU caches. The number of misses for an LRU set-associative cache are deduced as follows: let $p_i(s)$ be the probability that a reference is made to the ith most recently referenced cache line in one of the s sets, and let q_i be the probability that a reference is made to the ith most recently referenced cache line in any set. Consequently, $q_i = p_i(1)$. The miss ratio for an n-way set-associative cache with s sets is $1 - \sum_{i=1}^{n} p_i(s)$, while the miss ratio for a fully-associative cache with n lines is $1 - \sum_{i=1}^{n} q_i$. Bayes rule can be used to express the set-associative stack distance as follows:

$$p_n(s) = \sum_{i=1}^{\infty} Prob\,(\Delta = n \text{ with } s \text{ sets } \mid \Delta = i \text{ with 1 set }) \cdot q_i,$$

where Δ is the stack distance. Assuming that two cache lines map to the same set with probability $1/s$ and the mapping is independent from where other cache lines map, a reference to set-associative stack distance n occurs if exactly $n-1$ more recently referenced lines map to the reference's set. For a fully-associative cache, a stack distance of i implies $i-1$ most recently used lines. Thus, we can express $p_n(s)$ as:

$$p_n(s) \approx \sum_{i=1}^{\infty} \binom{i-1}{n-1} \left[\frac{1}{s}\right]^{n-1} \left[\frac{s-1}{s}\right]^{i-n} \cdot q_i, \text{ for } n \leq i.$$

This approximation has been proved to be quite accurate in [13].

3. APPLICATIONS

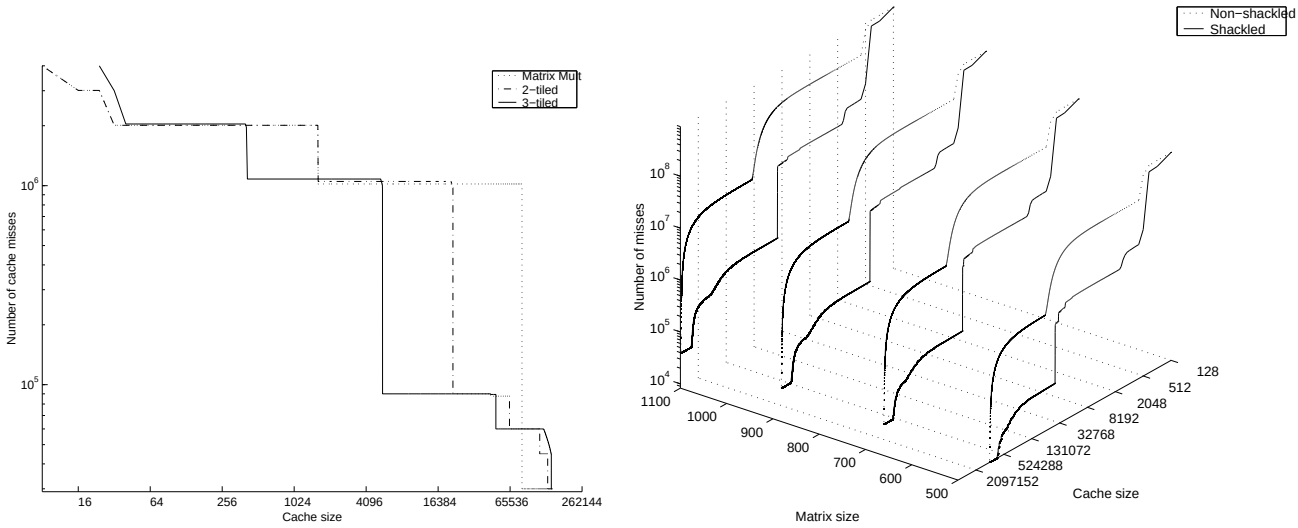
Once the stack histogram is computed symbolically, there are a number of applications, both within a compiler framework and outside of it. Inside the compiler the stack histogram can be used to guide optimizations – the user can retrieve the performance symbolic expressions and use computer algebra to compare the performance estimates of different code sections or versions. Outside the compiler framework, performance tools can use compile-time generated stack histograms to visualize performance and do scalability analysis by evaluating the symbolic expressions with different parameters.

In this section we present three applications of our compile-time algorithm for stack distances computation: machine independent evaluation of compiler optimizations with respect to data locality, tile size selection for loop tiling, and compile-time estimation of cache misses.

3.1 Machine Independent Evaluation of Compiler Transformations

To analyze the effectiveness of the stack histogram to capture data locality, we define a metric based on stack distances, and we select three compiler transformations that enhance locality: tiling, data shackling and the product-space transformation, and evaluate them using this metric.

Tiling [7, 33, 34], also known as blocking, is a transformation that increases the nesting degree of a loop, such that the working set of the innermost loop fits in the memory hierarchy level for which the transformation is applied. As an example, consider matrix multiplication $C = A \times B$. In each iteration of the innermost classic three nested loops



(a) Matrix multiplication

(b) Cholesky factorization

Figure 4: Cache misses as a function of the cache size and matrix size

implementation there are accesses to one row (or column) of C and A, and to the entire matrix B. If the matrices are large enough, data cannot be reused across iterations. We study two versions of tiling for matrix multiplication, 2-tiled – only the two outermost loops are tiled, and 3-tiled – all three loops are blocked. In Figure 4(a) we show the number of cache misses determined using Equation (1) as a function of the cache size for the three versions of matrix multiplication presented above. In this figure, the higher the number of references closer to the origin, the smaller the distance at which data is reused, therefore the better the locality and the lesser the number of cache misses.

Data shackling [14] optimizes locality by fixing a traversal order through the data structures of the program, and scheduling computations to be performed when a data item is touched. We evaluate the effect of data shackling on Cholesky factorization using the formula in Eq. (1) on a stack histogram collected at run-time. In Figure 4(b) we plot the estimated number of cache misses as a function of the matrix size and the cache size. We vary the size of the factored matrix from 500×500 to 1100×1100. The number of misses estimated for the shackled version is lower than the number of misses for the original code (the shackled curves are under the non-shackled ones). Although the number of misses increases with the matrix size, the shape of the histogram remains the same, showing once more that the stack histogram has little variance with respect to data size and indeed captures the access pattern in the code.

The product-space transformation [1] is a combination of loop fusion, distribution, and tiling, targeted to enhance the locality of imperfectly nested loops, such as Cholesky factorization, Jacobi relaxation and successive over-relaxation (SOR). These kernels are important in solving systems of partial differential equations. We show how this transformation affects our locality metric and the execution time of the kernels in Table 1.

Plotting the stack histogram (or the corresponding cache miss ratio) for a code segment gives a very intuitive view of the locality of that code segment – large numbers of references at small distances denote good locality. To abstract this view, we define a metric based on the area under the curve (see Figure 4), as follows:

1 Definition (Average locality) *Given a stack histogram S, with stack distances $\Delta = 1, \ldots, MAX$, and references $S(\Delta)$, the average locality is:*

$$AvgLoc = \frac{\sum_{\Delta=1}^{MAX} (\Delta \times S(\Delta))}{\sum_{\Delta=1}^{MAX} S(\Delta)} \quad (3)$$

In Table 1 we show the average locality computed for several kernels with different cache line sizes, before and after the transformations are applied. For each benchmark, two rows of data are presented, one for the original code and the second for the transformed code. The 32 bytes and 128 bytes cache line sizes correspond to 4 and 16 array elements per cache line, and are among the most commonly used values for cache line sizes in L1 and L2 caches, respectively. For each benchmark, the locality optimized benchmark has lower average locality, as expected.

3.2 Tile Size Selection

The performance of the tiled code critically depends on the selection of the tile size [32, 35]. In [35] the authors present an analytical model for estimating the parameters for tiling the matrix multiplication kernel. The equations for the optimal NB parameter (the tile size) that they derived manually can be automatically obtained from the stack histogram computed using our algorithm. The tile size is se-

Transformation	Benchmark	Input data size (block size)	Average Locality			Execution time (sec)
			Temporal	Spatial (CLS=32)	Spatial (CLS=128)	
Tiling	Matrix Mult	100	2619.890	221.146	46.971	0.265
	2-tiled MM	100(25)	886.451	93.087	30.014	0.254
	3-tiled MM	100(25)	367.794	30.219	6.026	0.241
Data Shackling	Cholesky (kij)	750	42474.783	2818.807	281.354	473.225
		750(64)	1678.441	108.808	11.267	86.667
Product-Space	Cholesky (ijk)	750	56550.634	2997.046	527.547	91.030
		750(40)	2280.934	152.073	49.426	86.381
	Jacobi	780	795652.422	42504.399	3481.207	2.904e-6
		780(20)	2163.203	144.745	15.073	1.904e-6

Table 1: Temporal and spatial average locality

lected such that it satisfies:

$$\Delta_i \leq C, \forall \imath, i \neq \infty,$$

i.e., all the stack distances must be less than the cache size.

3.3 Cache Miss Estimation

Using the stack histogram computed at compile-time, and the Equation (1), we can estimate the number of cache misses for a loop. In this section, we compare the number of misses computed using our compile-time algorithm with misses measured using hardware counters on the MIPS R10000 processor. The number of misses is an approximation because the stack models a fully associative cache while the R10000 has a 2-way set-associative cache, and we also consider one loop at a time in our analysis.

We have implemented our compile-time analysis stack processing instrumentation in the Polaris source-to-source compiler [3]. The compile-time algorithm computes the stack distances and the number of references at each distance as symbolic expressions. Then it generates code, such that the symbolic expressions can be evaluated for the input data size of the benchmark. The number of cache misses is estimated by plugging in the cache parameters in a post-processing step.

In Table 2 we compare the compile-time estimated cache misses with cache misses for the two data cache of the MIPS R10000 processor, measured using hardware counters. The cache characteristics are: 32 KB, 32 byte line size and two way associative for the L1 data cache and 1 MB, 128 byte line size, two way associative for the L2 unified data and instruction cache. The benchmarks in the table are a subset of the SPECfp95 benchmarks. For each benchmark, we estimated the number of cache misses for each loop nest that we can handle with our compile-time algorithm, and multiplied the estimation by the number of times the loop is executed.

4. RELATED WORK

There are many approaches targeted toward estimating the cache behavior of programs within a compiler framework. However, many factors, such as limited compiler information, algorithms complexity and hardware unpredictability, have made the problem so challenging that none of the proposed solutions is a complete solution.

Porterfield [26] presents one of the first static models of memory performance based on data dependences. For a loop, the *Overflow Iteration*, $O(i)$, is used to represent the maximum number of iterations that incurs no capacity misses. The overflow iteration provides a measure of how much data is accessed between the end points of a dependence. Ferrante et al. [9], use the same concept to estimate the number of cache misses at compile-time, computing the number of distinct cache lines accessed in a loop. They give exact formulas for the number of distinct array elements accessed when subscripts are functions of one or two loop variables, and provides an upper bound for a more general subscript functions. Ferrante's et al. approach appears to be less costly than Porterfield's, since the authors use the GCD test and Banerjee's inequalities instead of data dependence distance vectors. We can not readily compare the accuracy of our algorithm versus theirs because the experimental results presented in their paper are restricted to matrix multiplication, which both our algorithm and theirs predict with total accuracy.

Fahringer [8] presents an algorithm that estimates the number of cache misses for sequential and data parallel Fortran programs. The algorithm is based on the analysis of all array references in loop nests, classifying them with respect to data reuse and computing a cost function for the array classes that describes the cache behavior of the program. The author shows how to extend the cost function to procedures and entire programs, although no experimental results for entire programs are presented. There are two differences that make our algorithm more practical. First, it is not easy to see how Fahringer's algorithm can be extended to estimate inter-nest misses. And second, his algorithm needs the cache size as a parameter, while ours can estimate the number of cache misses for all the cache sizes based on the stack histogram. Again, it is very hard to see how effective is his algorithm compared to ours, because the only experiment presented in the paper is Jacobi relaxation, for which both algorithms are accurate.

McKinley[20, 21] uses a simple cost model for caches to drive optimizations for data locality and parallelism. In this model, references with group-spatial and group-temporal locality are grouped in equivalence classes using simple heuristics. The cost of a loop is given in terms of cache lines accessed by placing the loop as the innermost loop in the nest. Although the model is very approximate, it works quite well in practice, since in most cases it finds the correct relative ordering of the loops in the nest.

Ghosh, Martonosi and Malik [11] have introduced the *Cache Miss Equations* (CMEs) as a mathematical frame-

Benchmark	Loops		Cache Size = 32 KB		Cache Size = 1 MB	
	estimated	total	compiler	runtime	compiler	runtime
101.tomcatv	11	12	1031915250	1032772500	259715250	258208500
102.swim	24	24	1556623620	1547988763	404380548	391517395
103.su2cor	56	82	922260478	914107603	231318140	255751083
104.hydro2d	124	158	2108084122	2078098304	821557292	824698056

Table 2: Cache misses predicted at compile-time vs. measured using hardware counters on an MIPS R10000 processor. The two cache sizes correspond to the first and second level caches of the processor.

work that precisely represents cache misses in a loop nest. They estimate the number of cache misses in a code segment by counting the number of solutions of a system of linear Diophantine equations extracted from reuse vectors, where each solution corresponds to a potential cache miss. For each reuse vector, two kinds of equations are generated: *compulsory equations*, that represent cold misses, and *replacement equations*, which represent the interferences with other references. The number of cache misses is computed by traversing the iteration space and solving the system of equations at each iteration point. Although solving these linear systems is an NP-hard problem, the authors claim that mathematical techniques for manipulating the equations allow them to compute relatively easily the number of possible solutions without solving the equations. Our algorithm differs from theirs because in one single pass we can compute the stack histogram which can be subsequently used to estimate the number of cache misses for any cache size, thus avoiding the repeated execution of the expensive part of the algorithm. Vera et al. [30, 31] propose a solution based on sampling techniques to speed-up solving CMEs. Statistical sampling allows them to approximate the absolute miss ratio for each reference by analyzing only a small subset of the iteration space. Results are given with a confidence interval, parameterizable by the user. In [31] they expand this method to handle entire programs. However, they still do the analysis at loop level, and rely on inlining and code sinking to transform the entire program into one loop.

Chatterjee et al. [6] use Presburger formulas to express various kinds of misses as well as the state of the cache at the beginning and at the end of a loop. While exact, their method is exponential in the number of formulas to be solved, and thus, in the paper they present results only for matrix multiplication. For a matrix multiplication loop they require 108 cache miss formulas, which takes more than 241 seconds to solve. By comparison, our method takes about 3 seconds to generate the stack histogram for matrix multiplication.

5. CONCLUSIONS

In this paper we have presented our experience with stack processing algorithms. We have used the result of the algorithm, the stack histogram, to predict the cache behavior of scientific codes and to evaluate compiler optimizations for locality. The main advantage of using the stack histogram to analyze memory behavior is the cache size independence. The stack histogram is computed in one pass through the trace, and the program behavior can be analyzed for different memory or cache sizes working only with the histogram. The stack histogram can be computed at compile-time by analyzing the memory access pattern of the application. We have presented an algorithm that computes the stack histogram at compile-time based on data dependence distance vectors. While the algorithm has some limitations – it currently works at loop nest level for fully associative caches, we have shown that it can predict cache misses for a large number of loops in the SPECfp95 benchmark suite, with reasonable accuracy [5, 4].

We have integrated this work within the Delphi Project [27], where it is used to predict performance at compile time, as a tool for performance tuning and scalability analysis. The compile-time stack algorithm has also been used by Solihin et al. [29] to help schedule the code in an Intelligent Memory architecture.

Acknowledgments

We would like to thank Keshav Pingali and Nawaaz Ahmed for providing us with the code for data shackling and product-space transformations. This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract MIP-9619351; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

6. REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the International Conference on Supercomputing*, 2000.

[2] G. Almasi, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, June 2002.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, December 1996.

[4] C. Caşcaval, L. DeRose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In J. Ferrante and L. Carter, editors, *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[5] G. C. Caşcaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[6] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of PLDI '01*, pages 286–297, June 2001.

[7] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In

Proceedings of PLDI '95, La Jolla, CA, June 1995. SIGPLAN.

[8] T. Fahringer. Estimating cache performance for sequential and data parallel programs. Technical Report TR 97-9, Institute for Software Technology and Parallel Systems, Univ. of Vienna, Vienna, Austria, October 1997.

[9] J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness. In *4th International Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[10] 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(5):587–616, October 1988.

[11] S. Ghosh, M. Martonosi, and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of ASPLOS VIII*, San Jose, CA, October 1998.

[12] R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. *International Journal of Parallel Programming*, 25(2):113–146, April 1997.

[13] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[14] I. Kodukula, N. Ahmed, and K. Pingali. Data-Centric Multi-level Blocking. In *Proceedings of PLDI '97*, June 1997.

[15] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of PLDI '00*, pages 157–168. ACM Press, 2000.

[16] J. Lo, S. Eggers, H. Levy, S. Parekh, and D. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of 30th MICRO*, pages 114–124, December 1997.

[17] J. Lo, S. Eggers, H. Levy, and D. Tullsen. Compilation issues for a simultaneous multithreading processor. In *Proceedings of the First SUIF Compiler Workshop*, pages 146–147, 1996.

[18] N. Manjikian and T. S. Abdelrahman. Fusion of Loops for Parallelism and Locality. *IEEE Transactions of Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[19] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[20] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, March 1994.

[21] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, August 1998.

[22] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[23] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of PLDI '98*, 1998.

[24] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[25] P. M. Petersen and D. A. Padua. Experimental evaluation of some data dependence tests. Technical Report 1080, CSRD, 1991.

[26] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[27] D. A. Reed, D. A. Padua, I. T. Foster, D. B. Gannon, and B. P. Miller. Delphi: An integrated, language-directed performance prediction, measurement, and analysis environment. In *Frontiers '99: The 9th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, February 1999.

[28] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4(2):121–130, March 1978.

[29] Y. Solihin, J. Lee, and J. Torrellas. Automatically mapping code in an intelligent memory architecture. Submitted for publication.

[30] X. Vera, J. Llosa, A. Gonzales, and C. Ciuraneta. A fast implementation of Cache Miss Equations. In *8th International Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 319–325, Aussois, France, January 2000.

[31] X. Vera and J. Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 175–186, Feb. 2002.

[32] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. Technical Report UT CS-97-366, LAPACK Working Note No. 131, University of Tennessee, 1997.

[33] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of PLDI '91*, June 1991.

[34] M. Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, NV, November 1989. ACM.

[35] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. A. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of PLDI '03*, San Diego, CA, June 2003.