Calculating Reuse Distance from Source Code

Sri Hari Krishna Narayanan, Paul Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{snarayan,hovland}@mcs.anl.gov

***Abstract*—The efficient use of a system is of paramount importance in high-performance computing. Applications need to be engineered for future systems even before the architecture of such a system is clearly known. Static performance analysis that generates performance bounds is one way to approach the task of understanding application behavior. Performance bounds provide an upper limit on the performance of an application on a given architecture. Predicting cache hierarchy behavior and accesses to main memory is a requirement for accurate performance bounds. This work presents our static reuse distance algorithm to generate reuse distance histograms. We then use these histograms to predict cache miss rates. Experimental results for kernels studied show that the approach is accurate.**

***Keywords*-Computer performance; Performance analysis; Cache memory**

I. Introduction

As architectures evolve, application writers have to understand how to engineer their applications to run at peak performance. However, the value of peak performance derives its meaning from the algorithm implemented, application designed, and architecture used. From the architect's standpoint, peak performance is achieved when all the computational units are kept busy. From the algorithm's standpoint, asymptotic complexity is used to describe how quickly it will execute. Applications achieve high performance when an algorithm is implemented in a way that takes advantage of all the architectural features available. Often, this is far less than the theoretical peak of a machine. Performance bounds allow engineers to evaluate how close observed performance is to what is achievable.

PBound is a tool that combines application information and architectural information to generate realistic performance bounds for an application running on a particular system [**?**]. Given an input source code, PBound generates the closed-form expressions for floating point, load, and store requirements for the code to execute. Using these expressions it can predict an upper bound as a fraction of machine peak. However, it cannot predict wall-clock time accurately because the cache and memory behavior of the application is not captured by these expressions.

Cache misses can be classified as cold, capacity, or conflict. Cold or compulsory misses occur when a data element is loaded into the cache for the first time. Capacity misses occur because of the finite size of the cache, and conflict misses occur because of the mapping of blocks in main memory mapping to same position in the cache. This paper adds a reuse-distance-based memory model to PBound allowing it to predict cache hit/miss rates accurately. The algorithm acts on the source code to determine all the reuse patterns available in the code. This information can be combined with a variety of cache architectures to study the appropriateness of making an architectural design decision. However, the effect of loop transformations such as loop distribution and loop fusion on the reuse patterns in the code can be studied as well.

a b c d e d d b a f

Between the successive references to b
there are 3 unique references: c, d, and e

(a) Reuse distance for scalar references

```
for (i=0; i< N - M; i++)
    A[i] = A[i + M] * B[i];
```

Array B has no reuse. Reference
A[i] reuses data accessed by A[i+M]

(b) Reuse distance for array references

Fig. 1. Examples of reuse distance

Reuse distance is a measure of the number of unique data elements accessed between any two accesses to the same element. Fig. 1(a) presents an example of reuse distance for a set of scalar variables, and Fig. 1(b) shows the reuse distance present in a simple loop. For least recently used (LRU) caches, reuse distance can be used to accurately predict cache hit rates [**?**]. In an LRU cache, if the size of the L_l cache is d, all accesses with reuse distance less than d will hit in L_l or some cache L_m, where $m < l$.

The key contribution of this work is a static reuse distance algorithm the can be used with programs written in imperative programming languages such as C/C++ and Fortran. The algorithm detects accesses to both statically allocated and dynamically allocated arrays. By determining the volume of unique data between accesses to the same array location, the algorithm computes a reuse distance histogram for the program. The histogram allows the memory behavior of applications built as a series of loops in C/C++ and Fortran to be predicted. We have implemented the algorithm in PBound.

While there exist several accurate dynamic reuse distance

TABLE I
ACTIONS OF PBOUND WHEN UPON ENCOUNTERING NODES OF A PARTICULAR KIND IN THE AST.

Node Enountered	Action
Any Non-leaf Node	Node.Store $\leftarrow \sum_{ChildNode}$ ChildNode.Store Node.Load $\leftarrow \sum_{ChildNode}$ ChildNode.Load Node.Operation $\leftarrow \sum_{ChildNode}$ ChildNode.Operation
Variable Reference on LHS	Node.Store $\leftarrow$ Node.Store + 1
Variable Reference on RHS	Node.Load $\leftarrow$ Node.Load + 1
Operation	Node.Operation $\leftarrow$ Node.Operation + 1
Function Call	Node.Store $\leftarrow$ Node.Store + FCall.Store Node.Load $\leftarrow$ Node.Load + FCall.Load Node.Operation $\leftarrow$ Node.Operation + FCall.Operation
Loop	Node.Store $\leftarrow$ Loopbody.Store * IterationCount Node.Load $\leftarrow$ Loopbody.Load * IterationCount Node.Operation $\leftarrow$ Loopbody.Operation *IterationCount + Loopheader.Operation *IterationCount

estimation methods, the choice of a static algorithm fits into PBound's approach of avoiding the execution of code when possible. This also means that the time taken to determine the reuse distance for regular loops is independent of the size of the loop bounds. Experimental results show that the reuse model accurately predicts cache miss rates for key kernels.

This work cannot currently be used to predict miss rates for multithreaded applications, where one thread's accesses can cause the eviction of another thread's data from a cache. However, this work is a first step towards understanding multi-threaded behavior. Graphical processing units employ different a memory hierarchy compared with CPUs. Therefore, the predictions cache misses in this work are appropriate only for CPUs. However, CPUs with the cache hierarchies considered in this work are expected to continue to be used in future high-performance computing systems. Because this work works on source code, compiler optimizations that restructure the code can result in a different observed reuse distance. It is part of the planned future work of the code to consider code transformations and their effect on the reuse distance.

The rest of the paper is organized as follows. Section II discusses PBound, and Section III discusses related work. Section IV presents our reuse distance algorithm. Section **??** discusses low-level implementation details. Section **??** presents the experimental data. Section **??** discusses future work, and Section **??** presents our conclusions.

II. PBOUND

PBound [**?**] is a tool for generating performance bounds from source code. It *counts* the number of operations performed by a code with a simple architectural description to obtain the bounds on performance. PBound is built on the ROSE compiler framework [**?**], [**?**] and can generate bounds for codes written in C/C++ and Fortran. Given an input source code, PBound traverses the code's abstract syntax tree (AST) from the leaf nodes to the root. For every kind of node, the traversal generates generates closed-form expressions for floating point operations, integer operations, loads, and stores that can be attributed to a node. Table I summarizes the actions of PBound when it encounters different nodes of the AST. Consider a generalized *axpy* computation of the form $y = y + a_1x_1 + \cdots + a_nx_n$, where $a_1, \ldots, a_n$ are scalars and $y, x_1, \ldots, x_n$ are one-dimensional arrays. Fig. 2(a) shows the source code for $n = 4$. Fig. 2(b) shows the output generated for the source code. Fig. 2(c) and (d) present the parametrized expressions generated by PBound for the assignment statement and loop in the procedure. PBound can evaluate the possibility of using SIMD and fused operations and adjusts the expressions accordingly.

The generated code is a *slice* of the original computation and contains only the statements that are necessary for computing the bounds. The parameterized counts in either the log file or the transformed code can be evaluated either manually (by substituting appropriate values for all variables) or by simply compiling the generated source code and executing it in the same manner as the original application. The generated code contains a call to the `pboundLogInsert` function with parametrized expressions for the number of loads, stores, and arithmetic operations, with separate expressions for integer and floating-point values. A simple runtime library implements `pboundLogInsert` to keep track and output the loads, stores, and arithmetic operations. Because the AST traversal of the loop indicates that there are $4*n$ floating-point operations in the in loop, the generated code contains an expression indicating that there are $4*n$ floating-point operations (as well as other expressions). Since the generated code contains only a few simple expressions, it usuall takes much less time to execute than the original application. Thus, in many cases one can compute the performance bounds for a large application on much smaller resources.

PBound currently computes wall-clock time by using hypothetical cache hit rates for the kernel. As expected, these models do not match observed cache hit rates and thus lead to inaccurate execution times computation. While the counters for computation and load/stores computed by PBound are accurate for different architectures and kernels, predictions of wall-clock time are not accurate because the counters do not reveal whether the data is from a particular cache level or from main memory. The reuse distance model presented in this paper will be used to accurately predict the hit rates at

different cache levels.

III. Related work

This Section presents related work in the areas of reuse distance modeling and performance modeling.

A. Static Reuse Distance Models

Caşcaval and Padua [**?**] present an algorithm to estimate the number of cache misses by computing a stack histogram of reuse for loop kernels. To generate the stack histogram, they partition iteration spaces into regions with same incoming dependences. For each dependence they compute the intervening number of iterations and the array elements accessed within them. To compute the histogram, they consider each component of the iteration space, its incoming dependences, and the array elements for that dependence. Using the histogram, they are able to predict miss rates for associative and non-associative LRU caches. In contrast, our work does not rely on partitioning the iteration space. Therefore, we are are not limited to single loop nests.

Generating static reuse distance for Matlab code has been studied by Chauhan and Shei [**?**]. They present an efficient algorithm for computing reuse distances at the source level for whole programs, including inter-procedural data flow. For their source analysis, they assume that the code has been flattened, meaning that all expressions are broken down into their simplest form by introducing temporaries. Our work does not have this assumption because our anticipated users are unlikely to rewrite code for analysis and is hence this assumption a possible source of inaccuracy. Because we perform static analysis, in the future, we can use techniques such as common subexpression elimination automatically. Because their focus is on Matlab, where applications rely on libraries and vector operations and not deeply nested loops, their work is not directly usable for applications written in C/C++ and Fortran. Moreover, our reuse distance algorithm is written in a representation-independent format, we can apply it to a variety of languages. In contrast to [**?**], however, our work readily handles pointers, but cannot perform array section analysis.

Marin [**?**] presents a static binary analysis technique to derive symbolic formulae that describe the pattern of locations accessed by each memory reference. The formulae are used to instrument the binary code and obtain reuse distance histograms when the code is executed. Marin reasons that since predicting reuse distance at the instruction level is error-prone, executing the code is needed to account for data alignment in memory. In order to build a scalable reuse distance predictor, the author collects data from multiple executions with small data sets.

B. Dynamic Reuse Distance Computation

Ding and Zhong [**?**] have developed a dynamic reuse distance histogram generation method. They have built a tool called *LRUHistogramGenerator* based on Pin [**?**] to examine and instrument instructions that access memory locations. Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. When the instrumented instructions are executed, the memory locations accessed by them are recorded. A tree is used to organize the approximate last access time to memory locations and generate a reuse distance histogram. Because this method examines binary instructions, it is applicable to general code and can be readily used for whole program analysis. Furthermore, the authors show how predictions made for one input size can be scaled to other input data sizes. Our method on the other hand is more restrictive since it targets applications written as loops that access arrays inside them. As with any dynamic method, however, executing the application and analyzing instructions can take a long time which makes changes to the code expensive to re-analyze.

C. Performance prediction

Several commercial and research efforts exist to measure and visualize the performance of applications running on existing machines as well as simulators. Tools such as TAU [**?**] sample hardware counter information to build a performance profile of application execution. In contrast to these methods, PBound is used to predict the upper bound of performance of the application and does not rely on the execution of the application to characterize its behavior. The roofline model [**?**] is an intuitive visual performance model for multicore architectures to help identify which systems are a good match for important kernels as well as to changes to the kernel to improve their performance. The roofline model determines the upper bound on the performance of a kernel depending on the kernel's operational intensity which is determined empirically. In contrast to empirical measurement of the roofline model, PBound determines the upper bound of the kernel's performance statically.

IV. Reuse Distance Analysis

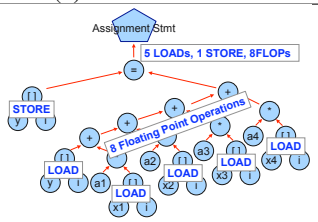
This Section presents the dataflow algorithms that have been implemented within PBound. The user invokes PBound as usual and in addition to the counter values generated earlier, PBound generates a reuse distance histogram. As will be explained in Section **??** the algorithms require several other tools as well. When PBound is invoked, a control flow graph (CFG) is created. The algorithms presented here operate on the CFG. In Section IV-A we will present the definitions of data structures within the data flow graph required to represent the loops and array references. Following this, in Section IV-B we associate each array reference with predecessor array references that it may reuse. In Section IV-C we show how the predecessors are ordered from nearest to furthest which determines the correct predecessor for every array reference. In Section IV-D we determine if an array reference can reuse its own data and if it has spatial reuse. Then, in Section IV-E we use the knowledge of predcessors and intervening statements in the AST to determine the reuse distance histogram. We use the matrix vector produce & transpose benchmark from

```
void axpy4(int n, double *y,
           double a1, double *x1,
           double a2, double *x2,
           double a3, double *x3,
           double a4, double *x4){
  register int i;
  for (i=0; i<=n-1; i++)
    y[i] = y[i] + a1 * x1[i] + a2 * x2[i]
                + a3 * x3[i] + a4 * x4[i];
}
```

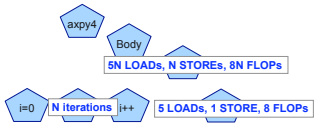
(a) Original code

```
#include "pbound_list.h"
void axpy4(int n, double *y,
           double a1, double *x1,
           double a2, double *x2,
           double a3, double *x3,
           double a4, double *x4){
  pboundLogInsert("axpy.c@6@5",
                  2, 0, 5*n+4,
                  n, 3*n+1, 4*n);
}
```

(b) Generated code



(c) Floating-point operations



(d) For loops

Fig. 2. (a) Example code, (b) generated output code with parametrized expressions,(c) traversal of assignment expression, (d) traversal of loop. The generated code indicates the following counts: integer loads: 2, integer stores: 0, floating-point Loads: 5*n+4, floating-point stores: n; integer operations: 3n+1; floating point operations: 4n.

the SPAPT benchmarks suite [**?**] given in Fig. 3 as a running example to explain the algorithms.

A. *Preliminaries*

The data flow algorithms require that the input code be represented by the following data structures given below. Fig. 4 presents the data structures for the example.

- Loop: A quartet of loop index, lower bound, upper bound, and step. $\{I_i, LB_i, UB_i, S_i\}$. The bounds must be numeric; a restriction that is addressed in Section **??**.
- Loop Nest LN: An ordered list $\{L_0 i,\ L_i,\ L_n\}$ of loops.
- Array Index Expression IE: An affine expression made up of at least one loop index.

```
#define N 2500
#define M 2500
int main(void) {
  int i, j;
  double a[M][N],y1[N],y2[M],x1[M],x2[N];
  for (i=0;i<=N-1;i++)
    for (j=0;j<=N-1;j++) {
    x1[i]=x1[i]+a[i][j]*y1[j];
    x2[j]=x2[j]+a[i][j]*y2[i];
  }
  return 0;
}
```

Fig. 3. Running example

LN_1: {{i, 1, N-1, 1},{j, 1, N-1, 1}.}
IS_1: {[i,j] : $0 \leq$ i $\leq$ 2499 && $1 \leq$ j $\leq$ 2499 }

AR_1: x1[i] $\rightarrow$ {x1, {i}}
DS_1:{IS_1, AR_1}: {[In1] : $0 \leq$ In1 $\leq$ 2499 }

AR_2: a[i][j]$\rightarrow$ {a, {i,j}}
DS_2:{IS_1, AR_2}:
{[In1,In2] : $0 \leq$ In1 $\leq$ 2499 && $0 \leq$ In2 $\leq$ 2499 }

AR_3: y1[j]$\rightarrow$ {y1, {j}}
DS_3:{IS_1, AR_3}: {[In1] : $0 \leq$ In1 $\leq$ 2499 }

AR_4: x2[j]$\rightarrow$ {x2, {j}}
DS_4:{IS_1, AR_4}: {[In1] : $0 \leq$ In1 $\leq$ 2499 }

AR_5: a[i][j]$\rightarrow$ {a, {i,j}}
DS_5:{IS_1, AR_5}:
{[In1,In2] : $0 \leq$ In1 $\leq$ 2499 && $0 \leq$ In2 $\leq$ 2499 }

AR_6: y2[i]$\rightarrow$ {y2, {i}}
DS_6:{IS_1, AR_6}: {[In1] : $0 \leq$ In1 $\leq$ 2499 }

$Stmt_1$: {AR_1,AR_2,AR_3}
$Stmt_2$: {AR_4,AR_5,AR_6}
Gen_{Stmt_1}: {DS_1,DS_2,DS_3}
Gen_{Stmt_2}: {DS_4,DS_5,DS_6}
$Proc_1$: {$Stmt_1$, $Stmt_2$}

Fig. 4. Data structures created

- Array Reference AR: A list of index expressions and an array. {A,{IE_1..IE_n}}
- Iteration Vector IV: An ordered list of values that the loop indices in the loop nest can assume.
- Iteration Space IS: The union of all possible iteration vectors.
- Data Space $DS_{IS,AR}$: The projection of the iteration space in the dataspace of the array in AR.
- Statement: The set of array references inside a loop.
- Gen_i: The set of DS in statement i.
- Procedure: An ordered list of loop nests.

$R_{AR_1 \to AR_1}$ = $\{DS_1 \cap DS_1\}$ = {[In1] : 0 $\leq$ In1 $\leq$ 2499 };
DV: {i-i} = {0}
$R_{AR_3 \to AR_3}$ = $\{DS_3 \cap DS_3\}$ = {[In1] : 0 $\leq$ In1 $\leq$ 2499 };
DV: {j-j} = {0}
$R_{AR_4 \to AR_4}$ = $\{DS_4 \cap DS_4\}$ = {[In1] : 0 $\leq$ In1 $\leq$ 2499 };
DV: {j-j} = {0}
$R_{AR_6 \to AR_6}$ = $\{DS_6 \cap DS_6\}$ = {[In1] : 0 $\leq$ In1 $\leq$ 2499 };
DV: {i-i} = {0}
$R_{AR_2 \to AR_2}$ = $\{DS_2 \cap DS_2\}$ =
{[In1,In2] : 0 $\leq$ In2 $\leq$ 2499 && 0 $\leq$ In1 $\leq$ 2499 };
DV: {i-i,j-j} = {0,0}
$R_{AR_5 \to AR_5}$ = $\{DS_5 \cap DS_5\}$ =
{[In1,In2] : 0 $\leq$ In2 $\leq$ 2499 && 0 $\leq$ In1 $\leq$ 2499 };
DV: {i-i,j-j} = {0,0}
$R_{AR_2 \to AR_5}$ = $\{DS_2 \cap DS_5\}$ =
{[In1,In2] : 0 $\leq$ In2 $\leq$ 2499 && 0 $\leq$ In1 $\leq$ 2499 };
DV: {i-i,j-j} = {0,0}

Fig. 5. Predecessor discovery and data reuse.

B. Predecessor Discovery

Algorithm 1 determines whether any two ARs in the code can reuse data. Reuse between a predecessor reference p and a current reference c is possible if c and p access the same array's data space. The data that is reused depends on whether c and p are in the same loop or not. If they are not in the same loop, the data that is reused is an intersection of the two data spaces. If c and p are in the same loop, then loop carried dependence should be considered. Formally, we define reuse between c and p to be the following.

$$R_{cp} = \begin{cases} \emptyset & \text{if } A_c \neq A_p \\ DS_{IS_c,AR_c} \cap DS_{IS_p,AR_p} & \text{if } LN_c \neq LN_p \\ DS_{IS_c,AR_c} \cap^* DS_{IS_p,AR_p} & \text{if } LN_c = LN_p \end{cases}$$

The operator $\cap^*$ is the loop-carried dependence operator. Loop-carried dependence is possible when DS_{IS_c,AR_c} is accessed by iteration vector IV_c, DS_{IS_p,AR_p} is accessed by iteration vector IV_p, $DS_{IS_c,AR_c} == DS_{IS_p,AR_p}$, and $IV_c \geq IV_p$. When $LN_c = LN_p$, we define the distance vector as $DV_{pc} = \forall_i IE_{ci} - IE_{pi}$.

Because data is reused from a predecessor and a successor, the first step in our algorithm is to associate each reference with every predecessor reference that could legally have reuse with it. Because, a reference could reuse itself through loop carried dependence, each AR is associated with itself as well. We use an iterative dataflow algorithm that traverses the nodes in the CFG. The iterative algorithm allows for the discovery of predecessors due to back edges in the CFG as well. For now, we limit ourselves to an intraprocedural algorithm because we do not expect function calls in the kernels that we are interested in. The output of the algorithm is a map between each AR and an ordered list of ARs whose data space it can reuse. Fig. 5 presents all the reuse that is output by the algorithm for our example. Although R_5 and R_6 are not truly possible, they are considered possible for now.

The algorithm can be described by three procedures. The first procedure, `ReuseAlgorithm`, iterates over the nodes in the CFG and visits each of them until their InSet and OutSet do not change. The InSet of a node is the set of nodes that are possible predecessors. The OutSet is the set of predecessors of the node and the node itself. Initially, all InSets and OutSets are empty.

The `VisitNode` procedure performs a join operation over all the incoming edges on the node and checks whether this is different from the existing InSet. If the InSet for a node has changed or if the node has never been visited before, the `Transfer` procedure for each statement in the node is called. If the output of `Transfer` is different from the OutSet of the node, the output is assigned to the OutSet. `Transfer` calculates the reuse possible between the references of the input statement and its predecessors and stores it in an ordered list.

```
for (i = 1; i < N; i++)
  x = a[i]; //Stmt1
for (i = 1; i < N; i++){
  y = a[i]; //Stmt2
  z = a[i]; //Stmt3
}
```

Fig. 6. The ordering of predecessors will ensure that the AR in Stmt3 will reuse the AR in Stmt2 and not the AR in Stmt3.

IS_1: {[i,j] : 0 $\leq$ i $\leq$ 2499 && 1 $\leq$ j $\leq$ 2499 }
AR_1: x1[i]: Self-temporal reuse in j; Spatial reuse: Yes
AR_2: a[i][j]: No self-temporal reuse; Spatial reuse: Yes
AR_3: y1[j]: Self-temporal reuse in i; Spatial reuse: Yes
AR_4: x2[j]: Self-temporal reuse in i; Spatial reuse: Yes
AR_5: a[i][j]: No self-temporal reuse; Spatial reuse: Yes
AR_6: y2[i]: Self-temporal reuse in j; Spatial reuse: Yes

Fig. 7. Spatial and Self-temporal reuse.

C. Ordering the Predecessors

The ordering of predecessors is an important task. For two predecessors, AR_{p1} and AR_{p2}, where $LN_c = LN_p$ we first consider the distance vector of the reuse. If $DV_{p1_c} \neq DV_{p2_c}$, then

$$\begin{matrix} Closer \\ Predecessor \end{matrix} = \begin{cases} c1 & \text{if } DV_{p1_c} < DV_{p2_c} \\ c2 & \text{if } DV_{p1_c} > DV_{p2_c} \end{cases}$$

If $DV_{p1_c} = DV_{p2_c}$ or $LN_c \neq LN_p$, we pick the reference with the fewer intervening statements/references. For the example in Fig. 5, there is only one instance of reuse that crosses statements. Therefore, ordering predecessors is not meaningful. However, for the example shown in Fig. 6 the ordering of predecessors will ensure that the AR in Stmt3 will reuse the AR in Stmt2 and not the AR in Stmt3.

D. Determining Spatial and Self-Temporal Reuse

A reference AR is determined to have spatial reuse in a loop nest L if the last index expression(IE) of AE is made up exclusively of a loop index L_l such as for any loop L_k where $k > l$ does not appear in any other IE in AR. A reference

Algorithm 1: Reuse Algorithm

Input: N: List of Nodes in the CFG
`ReuseAlgorithm`(N)
$InSet_{n \in N} \leftarrow \emptyset$
$OutSet_{n \in N} \leftarrow \emptyset$
forall the $n \in N$ **do**
  $Visited_n \leftarrow$ false
repeat
  $changed \leftarrow$ false
  foreach n *in* N **do**
    changed $\leftarrow$ changed && VisitNode(n)
until *changed* $\neq$ *true*;
return Reuse

`VisitNode`(Node n)
changed $\leftarrow$ false
tempInSet $\leftarrow \cup_{p \in predessor(n)}$
if $tempInSet \neq InSet_n$ **then**
  $InSet_n \leftarrow tempInSet$
  changed $\leftarrow$ true
if *changed* $\neq$ *false* $\vee$ $Visited_n \neq$ *true* **then**
  $Visited_n \leftarrow$ true
  changed $\leftarrow$ false
  tempOutSet $\leftarrow InSet_n$
  for *statement* $s \in N$ **do**
    tempOutSet $\leftarrow$ Transfer(tempOutSet, s)
  if $tempOutSet \neq OutSet_n$ **then**
    $OutSet_n \leftarrow tempOutSet$
    changed $\leftarrow$ true
return *changed*

`Transfer(inSet,s)`
Data: Def_s: Array references used in s
Data: Use_s: Array references defined in s
$Gen_i \leftarrow Def_i \cup Use_i$
foreach *Statement* $p \in$ *inSet* **do**
  $Reuse_{ip} \leftarrow (Gen_i \cap Gen_p$— $s.t.$ $\nexists k(Order(i,k) < Order(i,p))$
inSet $\leftarrow$ inset $\cup$ s
return *inSet*

AR is determined to have self-temporal reuse with loop L_l if L_l does not appear in any IE that makes up AR. While computing the reuse output in Algorithm 1, we keep track of whether the reuse has spatial reuse and whether self-temporal reuse is possible. Because of possible reuse with other ARs in a loop nest, the self-temporal reuse that is possible is not always manifest.

E. Computing the Reuse Distance Histogram

Algorithm **??** is used to compute the histogram. Its inputs are the results of Algorithm 1. For every pair of references AR_s and AR_d where AR_d reuses AD_s, the statements that occur in the node containing the AR_s and AR_d of the reuse as well as any intervening nodes are determined as a byproduct of Algorithm 1. Given a statement or set of statements, we define the operator $|s|_{[i] \rightarrow [i']}$ to be the size of the data space accessed by statement s for the loop iterations between iteration vectors $[i]$ and $[i']$. IV_s is the first iteration that accesses AR_s, and IV_d is the corresponding iteration that accesses AR_d.

Reuse can occur when AR_s and AR_d are in the same loop or in separate loops. Reuse can also occur when AR_s and AR_d are the same reference through self-temporal reuse. This case is handled by Algorithm **??**. If AR_s and AR_d are separate and in the same loop, the reuse distance is influenced by the partially executed loop iterations in which AR_s and AR_d are accessed. When AR_s and AR_d are not in the same loop, the reuse distance is the sum of data space accessed between IV_s and that occur before IV_d.

Algorithm **??** is used to determine the self-reuse of references that occur when the size of the IV is larger than the dataspace of AR. The algorithm iterates over surrounding loops; and based on whether the loops with self-temporal occur in the innermost position or the middle of the loop nest, it computes the number of times elements are reused and the reuse distance for this loop. The algorithm assumes that loops with reuse occur together and are not separated by other loops. This assumption can be overcome if the loop nest is rewritten to place loops with temporal reuse together in the loop nest.

Algorithm 2: Histogram Computation

`ComputeHistogram(ReuseOutput)`
foreach *Pair of references* (AR_s,AR_d) *in* $\in$ *ReuseOutput* **do**
  $S_s \leftarrow$ Statement containing AR_s
  $S_d \leftarrow$ Statement containing AR_d
  $B_s \leftarrow$ Statements in Node contaiging AR_s
  $B_d \leftarrow$ Statements in Node containing AR_d
  $B_sp \leftarrow$ Partial Statements containing AR_s
  $B_dp \leftarrow$ Partial Statements containing AR_d
  $IV_s \leftarrow$ First iteration vector for AR_s
  $IV_d \leftarrow$ First iteration vector for AR_d
  if AR_s *and* AR_d *are the same reference* **then**
    $Count_{sd} \leftarrow$ ComputeSelfTemporalReuseDistance(S_s)
  else if S_s *and* S_d *are in the same loop* **then**
    $Count_{sd} \leftarrow |B_{sp}|_{[IV_s]}$ + $|B_{sp}|_{[IV_s]}$ + $|B_{dp}|_{[IV_d]}$
  else
    N *gets* Intervening Nodes
    $Count_{sd} \leftarrow |B_{sp}|_{[IV_s \rightarrow I_{UB}]}$ + $|N|_{[I_{LB} \rightarrow I_{UB}]}$ + $|B_{dp}|_{[I_{LB} \rightarrow IV_d]}$
  Histogram[$Count_{sd}$] $\leftarrow |AR_s|_{[ReuseOutput]}$

V. Implementation

We have interfaced the existing PBound tool described in Section II with a representation-independent program analysis tool called OpenAnalysis [**?**]. OpenAnalysis uses the AST created by a compiler to generate a call graph and control flow

Algorithm 3: Compute SelfTemporalReuseDistance

```
ComputeSelfTemporalReuseDistance(S_s)
foreach SurroundingLoop l in SurroundingLoopNest L
do
    elementsreused ← 0
    reusedistance ← ∞
    lowerlooptimes ← 0
    loopitersize ← UB_l − LB_l
    looptimes ← 0
    if S_s does not have temporal reuse in l then
        if reusedistance==∞ then
            if elementsreused==0 then
                if looptimes == 0 then
                    looptimes ← (loopitersize -1);
                else
                    looptimes ← looptimes * (loopitersize
                    -1);
            else
                reusedistance ← elementsreused;
                elementsreused ← elementsreused *
                (loopitersize -1);
        else
            elementsreused ← elementsreused *
            (loopitersize -1);
    else
        if reusedistance==∞ then
            if looptimes == 0 then
                if elementsreused==0 then
                    elementsreused ← loopitersize;
                else
                    elementsreused ← elementsreused *
                    loopitersize;
            else
                reusedistance ← 0;
                elementsreused ← looptimes *
                loopitersize;
                looptimes ← 0;
        else
            elementsreused ← elementsreused
            *loopitersize;
Histogram[reusedistance] ← Histogram[reusedistance] +
elementsreused
Histogram[∞] ← Histogram[∞] + |L| - elementsreused
```

```
#define N 2500
  for (i=0;i<=N-1;i++) {
    temp1 = x1[i];
    temp2 = y_2[i];
    for (j=0;j<=N-1;j++)
    {
      temp3 =a[i][j];
      temp1=temp1+temp3*y_1[j];
      x2[j]=x2[j]+temp3*temp2;
    }
    x1[i] = temp1;
  }
```

(a) Modified mvt

```
#define STREAM_ARRAY_SIZE 1000000
for (j=0; j<STREAM_ARRAY_SIZE; j++)
  a[j] = b[j]+scalar*c[j];
```

(b) Stream triad

```
#define N 350
for (i = 0;  i < N; i++){
  for (j = 0; j < N; j++){
    b[i][j] = a[i][j] -1;
   }
}
for (i = 0;  i < N; i++){
  for (j = 0; j < N; j++){
    y[i][j] = a[i][j];
  }
}
```

(c) loop fission

```
#define N 350
for (i = 0;  i < N; i++){
  for (j = 0; j < N; j++){
    b[i][j] = a[i][j] -1;
    y[i][j] = a[i][j];
  }
}
```

(d) loop fusion

```
#define N 1024
#define M 32
double C[N][N], A[N][M], B[M][N];
for (i = 0; i<N;i++){
  for (j = 0; j<N;j++){
    temp = 0;
    for (k = 0; k<M;k++){
      temp +=  A[i][k] * B[k][j];
     }
     C[i][j] = temp;
  }
}
```

(e) mxm long-and-skinny

Fig. 8. Benchmarks studied

graphs and performs alias analysis, DUUD chains, side-effect analysis, liveness analysis, activity analysis and linearity analysis. Using the existing infrastructure for dataflow analysis, we implemented our static reuse distance algorithm that generates information regarding the reuse of array data structures. Implementing the algorithm in a representation independent format allows it to be used by tools other than PBound. The reuse distance algorithms are invoked automatically when PBound is invoked on the input code.

Gen_i is calculated by the polyhedral analysis tool OmegaLibrary [**?**] using the indices that are present in the array

references and the loop bounds. OmegaLibrary is also used to perform the union, intersection, and difference operations on different data spaces. SymPy is a Python library for symbolic mathematics [**?**]. It was used to manipulate the array index expressions symbolically and compute distance vectors.

VI. Experimental Setup

We use the five benchmarks shown in Fig. **??** to evaluate the algorithm. For the mxm benchmark, we can exchange the values of N and M to make the array either long and skinny or short and fat. The mvt benchmark is the running example that we have used in the code, with modifications that a compiler makes. We discuss the possibility of automatically performomg code transformations in Section **??**.

In order to validate the reuse histogram generated by PBound, we compare it against the histogram generated by LRUHistogramGenerator [**?**]. LRUHistogramGenerator is built on the binary instrumentation tool Pin. It instruments instructions that refer to memory locations. When the instrumented instructions are executed, the memory locations accessed by them are recorded, and a reuse distance histogram is generated. By default, LRUHistogramGenerator generates a histogram for the whole application binary including the operating system instructions to start the application. Additionally, LRUHistogramGenerator counts only the first access to a packed (64-bit, 128-bit) memory location as a unique access. Subsequent accesses to members of the packed location are counted as reuse. To enable validation, we have made very minor modifications to this default behavior and refer to this modified tool as LRUHistogramGenerator* in the rest of the paper. While it would be possible to modify PBound to obtain the same values as LRUHistogramGenerator, it would have required an unacceptable change to the algortihms that PBound is build upon. Moreover, the use case for PBound relies on the unmodified histogram values that PBound generates.

Fig. **??** compares the reuse distance histogram generated by PBound and LRUHistogramGenerator* for the four benchmarks. One can see that for all benchmarks PBound and LRUHistogramGenerator* match well. Where the graph shows differing reuse distance values, the reason that a small difference in the reuse distance value close to the boundary of a bin size can place entries in different bins. For the mvt benchmark, one sees that for the smallest reuse distance (64 bytes), LRUHistogramGenerator* has a high number of occurrences, while PBound has next to none. This is because PBound *currently* does not consider the reuse of scalar variables while LRUHistogramGenerator* does include them in its counts. Because scalar variables are held in registers, it will be shown below that overlooking the reuse distance occurrences caused by them will not cause a difference in the cache miss rate.

A. Cache Hit Rate Results

Using the histograms generated for the different benchmarks, we generate cache miss rates for an LRU cache hierarchy where each cache is fully associative. We compare these against the cache simulator provided by the PIN tool *allcache* [**?**]. While measurements can be made against a real system, we find that the hardware prefetcher reduces the observed miss rates. We plan to incorporate the effect of prefetching at a later point, as discussed in Section **??**. Allcache was modified to instrument only those memory access instructions that occur inside the function of interest. This allowed us to ignore accesses attributed to sources such as the loading of the application binary and initialization routines. While such accesses are of interest in contexts where the entire application's execution is considered, they are not of interest in validating PBound's predictions. Allcache allows us to describe a cache hierarchy of arbitrary depth.

TABLE II
Details of cache hierarchy simulated

Level	Size	Line Size	Line Count	Associativity	Store policy
L1	32 KB	64 bytes	512	Fully	Store Allocate
L2	16 MB	64 bytes	262144	Fully	Store Allocate

Cache sizes, associativity, and line sizes can be specified. Caches can also specify whether stores should result in allocation. Instruction caches, data caches, and unified caches can be modeled by appropriate specification of the hierarchy and instrumentation routines. When each instrumented instruction is executed, the memory address is searched for in the cache hierarchy, and a hit or a miss at the appropriate cache levels is determined. Table **??** outlines the cache hierarchy we employed in our experiments. Based on the histogram, we can predict that any reuse with distance greater than L1 cache size will miss in L1 and any reuse with distance greater than L2 cache size will miss in L2. We note, however that once a cache block is loaded into the cache, spatial reuse will affect the miss rate. Therefore, we consider spatial reuse as well while making our predictions. Based on this approach we compare the miss rate predicted by PBound and that observed by allcache for the different benchmarks. Fig. **??** shows that there is a high agreement between the two. However, for mxm long-and-skinny, PBound's predictions for L1 miss rates are much higher than the rate determined by allcache. The reason is that PBound determines that there exists no spatial reuse in array B whereas allcache sees spatial reuse. This may be because of compiler transformations that restructure the loop; further investigation is merited.

VII. Future work

In this Section we address the work required to improve the accuracy and applicability of the algorithms.

A. Improved Anaysis

We would like to analyze codes with symbolic loop bounds. Currently they are restricted to numeric bounds. While the descriptions of the array references and data spaces in Algorithm 1 will remain the same, additional constrains that bind the symbolic bounds to numerical values will generate numeric reuse results without changing the source code. To study the effect of tiling, we will have to add the ability to introduce additional loops and express the bounds of one

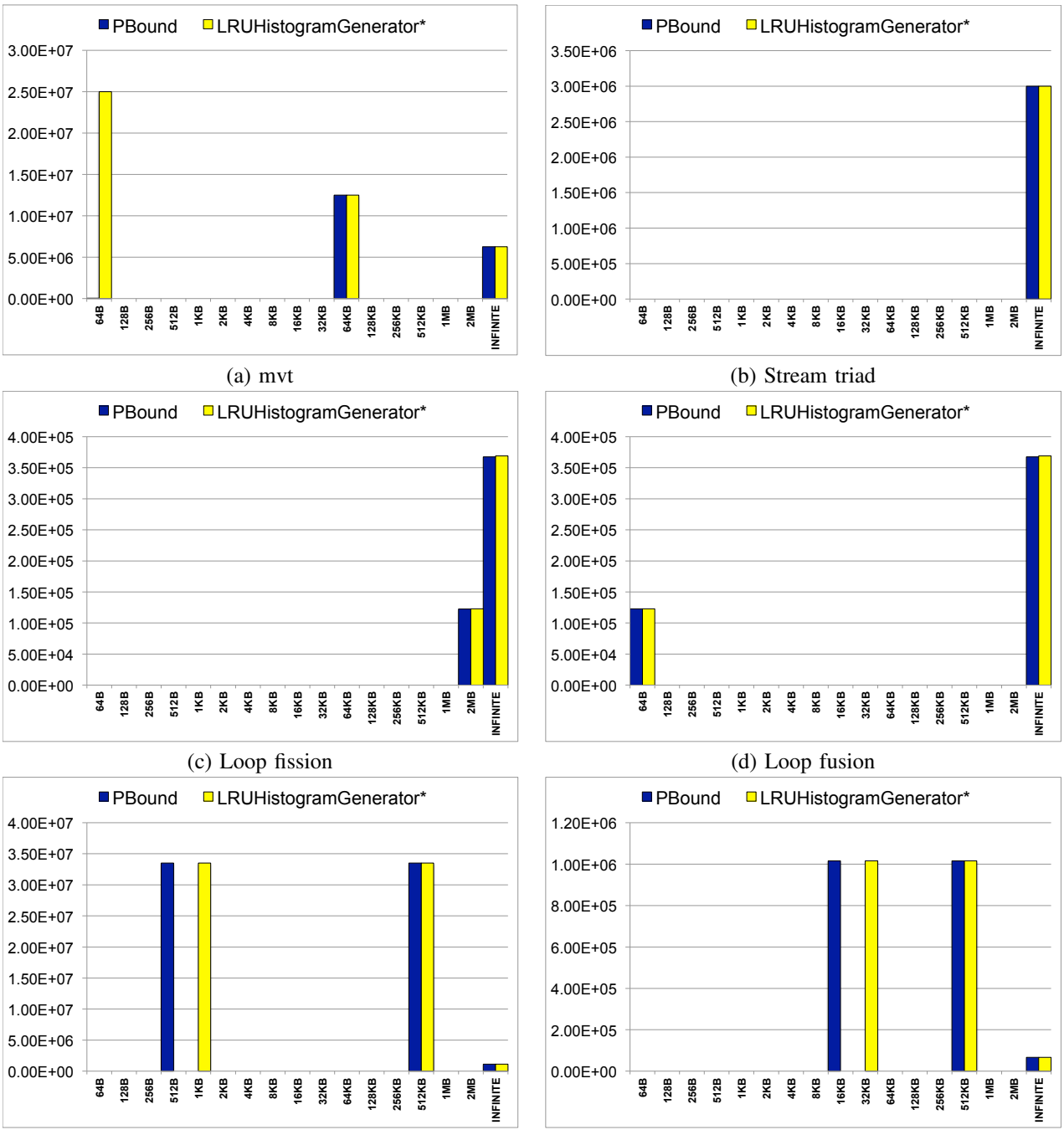


Fig. 9. Comparison of static and dynamic reuse distances. The x-axis is the reuse distance in bins of size starting at 64 bytes. The last bin indicates that there is no reuse. The y-axis is the number of instances of reuse at a particular reuse distance.

loop as an affine function of another loop. We would also like to handle sparse data accesses by using user provided descriptions of sparse data layout. Currently, changes to the code require a complete reanalysis of the source code. The reason is that objects such as array references are linked implicitly to the Rose AST node that they are formed from. We will study whether the intermediate values of code analysis can be represernted in a format that breaks this linkage, so that the values can be changed on demand.

B. Bound Generation

We will extend and validate our algorithms for non-fully associative cache hierarchies. We will combine the prediction of cache miss rate with the values gained from counting the number of floating-point operations to compute upper bounds on application performance and compute wall-clock execution time. When the bounds and execution times are validated for observable machines, we will explore future architectures that

may be expected in the realm of high-performance computing. Exascale machines are expected to have reduced memory and high communication cost. Using PBound to predict the utilization of different memory configurations is one way to evaluate the performance of applications on these future machines.

C. Use of Prefetching

In order to utilize the reuse distance algorithm to predict misses in current and future machines, the behavior of the hardware prefetcher will have to be included. The hardware prefetcher works by recognizing streaming access patterns. Marin et al. [**?**] describe an approach based on static analysis and simulation in order to understand whether the memory access patterns of applications are amenable to hardware prefetching and to identify opportunities for improving their prefetch friendliness. Incorporating such an approach will allow us to statically determine the memory behavior using only source code.

D. Models to Drive Autotuners

Autotuning is a method of generating semantically equivalent multiple versions of the same code. The automated versions are generated from specifications of compiler transformations for the code. By generating these different versions and empirically measuring their performance, the best version can be selected. The transformations can be guided by the reuse distance model in order to select the loops to unroll, the unroll factors and the references to prioritize. Furthermore, while empirically evaluating the different versions, the bound generated by PBound can guide the empirical evaluation by presenting a criterion for stopping the search process once a variant that is close the bound is found.

VIII. Conclusions

We have created a new static analysis based reuse distance algorithm that analyzes the source code of a program written in an imperative programming language such C/C++ or Fortran and generates reuse distance histograms. The algorithm has been implemented in PBound and the histograms generated by our reuse distance algorithm match the histograms determined by a dynamic reuse distance calculation method. The generated histograms have been used for estimating the cache hit rates for a CPU cache hierarchy. The cache hit rates computed by a binary analysis method match those computed by using our reuse distance algorithm.

ACKNOWLEDGEMENT

This work was funded by a grant from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357

References

[1] Intel PIN web page. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[2] SymPy web page. http://sympy.org

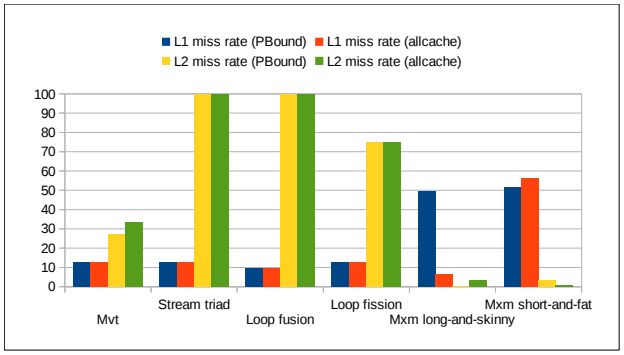


Fig. 10. Comparison of cache miss rates predicted by PBound and computed by PIN's allcache tool.

[3] Balaprakash, P., Wild, S.M., Norris, B.: Spapt: Search problems in automatic performance tuning. Procedia Computer Science 9(0), 1959 – 1968 (2012), http://www.sciencedirect.com/science/article/pii/S1877050912003353, proceedings of the International Conference on Computational Science, {ICCS} 2012

[4] Casçaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: Proceedings of the 17th Annual International Conference on Supercomputing. pp. 150–159. ICS '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/782814.782836

[5] Chauhan, A., Shei, C.Y.: Static reuse distances for locality-based optimizations in MATLAB. In: Proceedings of the 24th ACM International Conference on Supercomputing. pp. 295–304. ICS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1810085.1810125

[6] Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. pp. 245–257. PLDI '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/781131.781159

[7] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Tech. rep., College Park, MD, USA (1995)

[8] Marin, G.: Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In: Proceedings of the Symposium of the Las Alamo s Computer Science Institute, Sante Fe, NM, USA (2005)

[9] Marin, G., McCurdy, C., Vetter, J.S.: Diagnosis and optimization of application prefetching performance. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 303–312. ICS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2464996.2465014

[10] Narayanan, S.H.K., Norris, B., Hovland, P.D.: Generating performance bounds from source code. In: Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010). pp. 197–206 (2010), also available as Preprint ANL/MCS-P1685-1009

[11] Quinlan, D.: ROSE web page. http://rosecompiler.org

[12] Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In: JMLC'03: Joint Modular Languages Conference. Lecture Notes in Computer Science, vol. 2789, pp. 214–223. Springer Verlag (Aug 2003)

[13] Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (May 2006), http://dx.doi.org/10.1177/1094342006064482

[14] Strout, M.M., Mellor-Crummey, J., Hovland, P.: Representation-independent program analysis. In: Proceedings of the sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE) (Sep 5-6 2005)

[15] Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (Apr 2009), http://doi.acm.org/10.1145/1498765.1498785