Virtual Reuse Distance Analysis of SPECjvm2008 Data Locality

Xiaoming Gu
Intel China Research Center
xiaoming@cs.rochester.edu

Xiao-Feng Li
Intel China Research Center
xiao.feng.li@intel.com

Buqi Cheng
Intel China Research Center
bu.qi.cheng@intel.com

Eric Huang
Intel China Research Center
eric.huang@intel.com

ABSTRACT

Reuse distance analysis has been proved promising in evaluating and predicting data locality for programs written in Fortran or C/C++. But its effect has not been examined for applications in managed runtime environments, where there is no concept of memory address. For this reason, traditional reuse distance analysis based on memory addresses is not directly applicable to these applications.

This paper presents the *Virtual Reuse Distance Analysis* (ViRDA), which resolves the difficulties associated with runtime environments and provides insights into the high-level locality in dynamic applications. ViRDA addresses the problem caused by managed runtime artifacts, garbage collection in particular, by using virtual data identities, obtained through a standard profiling interface, to capture inherent data locality. The effectiveness of ViRDA is evaluated using a subset of the SPECjvm2008 benchmark suite. The new analysis reveals the reuse distance signatures of these programs and helps to explain the cause of excessive cache misses. It also predicts locality for large inputs based on training analysis of several small inputs. The prediction error is no more than 6% for the 4 *scimark* workloads.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, runtime environments*; C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques*

General Terms

Measurement, Performance

Keywords

Data locality, Reuse distance, Managed runtime, SPECjvm2008

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.

1. INTRODUCTION

A lasting phenomenon in computing technology is the ever growing gap between CPU and memory, which imposes a heavy penalty at cache misses and gives rise to one of the most severe performance bottlenecks. At the same time, applications based on managed runtime systems are gaining popularity due to their programmability, portability, and availability. To minimize the memory penalty for these dynamic applications, we need ways to analyze their data locality.

Reuse distance [16, 7, 18, 11], as a precise quantitative metric for modeling locality, has been used extensively. However, previously existing tools typically deal with programs written in native languages such as Fortran or C/C++. Applications written in these languages are compiled directly to binaries. Since memory objects do not change location after they are allocated, there is direct correspondence between an object and its memory address. This property does not hold in applications executing in managed runtime environments such as Java, C#, and Java Script, where data could be relocated due to garbage collection (GC). In this scenario, an object can potentially reside in one memory location before a GC and another location after the GC. Furthermore, the activities to a virtual machine (VM) incur a large number of memory accesses that are not directly related to the actions of applications. For example consider the effect of a just-in-time compiler (JIT). These artifacts make it difficult to analyze the locality of dynamic applications using the traditional approach. A new mechanism is needed for these applications.

This paper presents a new mechanism called *Virtual Reuse Distance Analysis* (ViRDA). ViRDA works with virtual data identities, which removes the impact of VM artifacts, especially GC. By observing memory accesses virtually using symbolic identities instead of actual addresses, we can capture the inherent data locality of an application. Given the application and its input parameters, its locality profile is largely machine-independent and VM-independent. In other words, the locality information is *profile-once-use-everywhere*, which follows the *write-once-run-everywhere* spirit of virtual-machine applications.

The paper shows the use of ViRDA on a carefully selected subset of SPECjvm2008 [6] programs. The tool measures the reuse signatures (explained in Section 2.2), detects working set sizes, and helps to analyze long-distance reuses. In addition, it is used to predict reuse signatures for large data inputs based on training analysis of the reuse signatures from

executions using small inputs.

The rest of this paper is organized as followings. Section 2 introduces the basic definitions and related work. Section 3 explains the detailed design of ViRDA. Section 4 presents a series of locality profiles for the selected SPECjvm2008 benchmarks. Finally, Section 5 discusses the limitations of ViRDA and presents the future work, and Section 6 summarizes the paper.

2. BACKGROUND AND RELATED WORK

The memory wall [17], the growing gap between CPU and memory, is one of the most severe performance bottlenecks. Data locality has been under active study for decades for ameliorating the effect of the memory bottleneck.

2.1 Data locality

There are two aspects of data locality:

- Temporal locality—the currently accessed data element will be accessed again in the near future;
- Spatial locality—the data elements adjacent to the currently accessed element will be accessed in the near future.

Techniques such as computation reordering [10, 14] and data reorganization [9, 26, 19] are commonly used to improve data locality. Typically the benefits of optimizations are evaluated by real performance measurements, which is dependent of hardware and operating system (including tuning parameters). A metric such as reuse distance to model data locality may help to separate application-specific factors from machine- and system-specific effects.

2.2 Reuse distance and reuse signature

The *reuse distance* (RD) for a memory access is the number of distinctive data elements between this access and the previous access to the same data element. It was first defined in 1970 by Mattson et al. as one of the stack distances in their seminal study of locality for virtual memory systems [16]. An example is shown in Table 1. Intuitively, reuse distance analysis simulates a fully associative cache with least recently used (LRU) replacement policy and infinite capacity. If a data element is never accessed before, the reuse distance of the first access is infinite. In practice, an access with a shorter reuse distance is more likely to result in a cache hit.

Access trace	a	b	c	a	b	b	a	c
Reuse distance	∞	∞	∞	2	2	0	1	2

Table 1: Example reuse distances

To represent all reuse distances in a data access trace, the *reuse signature* was introduced, which is the distribution of all reuse distances [11]. The reuse signature for the example in Table 1 is shown in Figure 1, assuming that each data element occupies one byte. Reuse distances are grouped into bins with widths in base two logarithmic scale. For a reuse signature with N bins, bin 0 is for distance 0, bin 1 for distance 1, and bin n ($2 \leq n \leq N-3$) for reuse distances between $[2^{n-1}, 2^n - 1]$, bin $N-2$ for finite reuse distances no less than 2^{N-3} and the last bin for infinite distance (the data is never accessed before). Locality optimizations aim to shorten long distance reuses, and their improvements can be measured by comparing the reuse signatures before and after the optimization.

Figure 1: An example reuse signature

2.3 Related Work

The time complexity of the algorithm by Mattson et al. is $O(NM)$ and its space complexity is $O(M)$ since it simply simulated a fully associative LRU cache [16], where N is the number of memory accesses and M is the number of data elements. Bennet and Kruskal computed reuse distance with a partial sum hierarchy to achieve time complexity $O(NlogN)$ and space complexity $O(N)$ [7]. Olken improved the work by using an AVL tree and its time and space complexities were lowered to $O(NlogM)$ and $O(M)$ respectively [18]. Ding and Zhong invented an approximation algorithm based on a scale tree (a splay tree [24] with dynamic compression) and achieved time complexity of $O(NloglogM)$ and space complexity of $O(loglogM)$ [11]. Our tool uses two of the algorithms: the Olken algorithm with a splay tree [18] and the approximate algorithm with a scale tree [11].

Time distance (or access distance [12]) is the number of accesses between two accesses to the same data element without concerning with distinctiveness. For example, the time distance for the last access in Table 1 is 4. Compared with reuse distance, time distance is much easier to compute. Shen et al. used time distance to approximate reuse distance [21, 20]. Beyls and D'Hollander applied reservoir sampling [15] in conjuction with time distance [8]. Zhong and Chang proposed a history-preserving representative sampling for accelerating reuse distance analysis [25].

Ding and Zhong proposed to predict locality based on reuse signatures for large inputs using the results from small inputs [11]. Shen et al. used reuse distance as a basic metric to detect locality phases at run time and mark their boundaries in source code [22]. With reuse distance analysis, Fang et al. found out a set of critical instruction accounting for most of the L2 cache misses in both integer and floating-point programs [12]. Beyls and D'Hollander built a tool named Suggestions for Locality Optimizations (SLO)[8], which could find code with temporal locality problems for native applications. Gu et al. studied spatial locality based on the changes of reuse distances when doubling the granularity of data blocks [13].

Shiv et al. conducted a thorough study of SPECjvm2008

on Intel platforms using hardware event statistics [23]. But the locality behavior is not characterized.

All prior work with reuse distance is for native applications, where data accesses can be identified by memory addresses. However, these mechanisms do not work well with managed runtime systems. There are interfering memory references by the runtime system internals, i.e., the VM but mostly irrelevant or not directly related to the locality profile of the application itself. To make things worse, copying-based garbage collectors move objects around in memory frequently. If memory addresses are used, the same object could be regarded as multiple different data items. Next we describe our solution to these two problems, a solution based on virtual data identities.

3. VIRTUAL REUSE DISTANCE ANALYSIS

Symbol-based analysis has two significant advantages compared with memory address based analysis:

- It only captures the data accesses made by applications;
- It is not affected by data relocation due to GC.

With virtual object identity, application locality profile can be captured because only data accesses of the application are considered by the analysis. Local variables are allocated in registers or on stack, and they typically have very short reuse distances that result in cache hits, making them uninteresting for locality analysis. Hence, our focus is on heap data accesses. We call this symbol-level reuse distance analysis as *Virtual Reuse Distance Analysis*, or *ViRDA* for short.

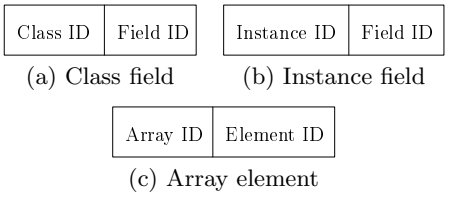
In the rest of this paper, we present ViRDA in the context of the Java virtual machine, although the technique can be generally applied to other managed runtime systems.

3.1 Virtual Data Identity

In ViRDA, heap objects are categorized as follows:

- Class field—Class ID (class signature combined with class loader) and field ID are used to represent a class field in Figure 2(a). The field ID is the same as field index in Java class file.
- Instance field—Instance ID and field ID are used to represent an instance field. The instance ID equals to the hash code returned by *System.identityHashCode()*, which remains unchanged throughout the lifetime of an instance. See Figure 2(b) for details.
- Array element—Array ID and element ID are used to represent an array element in Figure 2(c). An array is just a special object and it has a hash code as its ID. The element ID is the element index.

Using identity traces from data access profiling, we can study the symbol-level locality for an application. In Figure 3, we show an example using virtual reuse distance with code snippets in Figure 3(a) and Figure 3(b), and the corresponding resulted access trace in Figure 3(c). The reuse distance computation is done by a reuse distance analyzer discussed in Section 3.2.2. Here *virtual reuse distance* (ViRD) is measured in bytes. As shown in Figure 3, ViRD is augmented from RD with data size information. For instance,



(a) Class field (b) Instance field

(c) Array element

Figure 2: The design of virtual data identity

the second access to *b.ctr* has two data elements of 4 bytes referenced in between, then the ViRD for this access is $4 + 4 = 8$. We will use ViRD instead of RD in the rest of this paper, but we simply refer to it as reuse distance (RD).

```
class Apple() {}
class Peach() {}
class Basket() {
    long ctr = 0;
    Apple a = null;
    Peach p = null;
}

Peach oPeach1 = new Peach();
Peach oPeach2 = new Peach();
Apple oApple1 = new Apple();
Apple oApple2 = new Apple();
```

(a) Data initialization

```
Basket b = new Basket();

b.p = oPeach1;
b.ctr = 1;

b.a = oApple1;
b.ctr = 2;

b.a = oApple2;
b.p = oPeach2;
```

(b) Data access

Access trace	b.ctr	b.a	b.p	b.p	b.ctr	b.a	b.ctr	b.a	b.p
RD	∞	∞	∞	0	2	2	1	1	2
Data size	8	4	4	4	8	4	8	4	4
ViRD	∞	∞	∞	0	8	12	4	8	12

(c) Virtual reuse distances for the accesses in (b)

Figure 3: Example virtual reuse distances

The example above assumes that each field is mapped to a distinctive cache block. (Here a field refers to a field of a class, an instance, or an element of an array.) In practice, a field typically takes less than a cache block, and multiple fields may occupy the same cache block. To reflect this, we use "virtual allocation" for each field and assign the field a *data block ID* as follows:

- Assume each object is allocated starting from a cache block aligned location, where the data block ID is recorded as 0.
- Allocate each field of the object in the order of declaration in the current cache block; use the next cache block if the remaining space in current block is inadequate; record the cache block number as its data block ID.
- Add data block ID to every field identity as shown in Figure 4 below.

The reuse distance analysis based on identities with data block ID shows block temporal locality, which includes block-internal spatial locality. In comparison, the analysis based on the identity design in Figure 2 is for pure temporal locality.

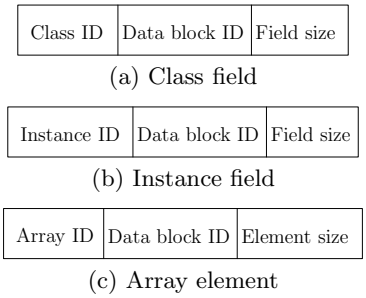


Figure 4: Virtual data identity with data block ID

3.2 A JVMTI-based Implementation

With the data identity design, ViRDA is implemented using the JVM Tool Interface (JVMTI) [4] for Java applications. It consists of two parts:

- Front end—profiling the trace of virtual data access;
- Back end—computing reuse distances.

3.2.1 The Front End

ViRDA catches data accesses of Java applications using the standard JVMTI [4]. JVMTI is an event-based programming API for instrumenting and profiling Java applications.[1] The virtual data identities are constructed in the formats shown in Figure 2 and Figure 4. The identities are put into a dedicated buffer, which transfers profiling data from the front end to the back end.

3.2.2 The Back End

The back end is a reuse distance analyzer to process the data buffer when it is full. It computes reuse distance for each virtual access in the buffer. When buffer processing is finished, it waits for JVMTI event handlers to fill the buffer again. Currently, there are two analyzers available. The exact analysis is based on a splay tree with time complexity $O(NlogM)$ and space complexity $O(M)$ and the approximate one is based on a scale tree with time complexity $O(NloglogM)$ and space complexity $O(logM)$. The approximate analyzer is much faster than the precise one, so we use the faster analyzer when analyzing larger applications. The accurate analyzer is used for working set prediction, which gives clear component information even for bins with very short reuse distances. A code sketch of the accurate analyzer is shown in Figure 7. The splay tree as a binary tree organized by the access order and a field of tree node records the sub-tree size including the node itself. The algorithm for scale tree has a similar framework except that it uses dynamic compactions to integrate adjacent nodes into a single node when measurement precision is not affected [11].

After a reuse distance is calculated, it is recorded in the corresponding distance bin. After profiling, the reuse signature is plotted using some external tools. Besides reuse signatures, hot methods, classes and fields with long distances are reported with additional bookkeeping, which helps to identify locality problems in the source code. Developers and performance tuners can benefit from such information.

[1]Some necessary JVMTI extensions are added in order to obtain all data identities. For instance, obtaining array accesses, getting array hash code, getting element index, and getting field offset.

```
Take a virtual access from the data buffer;
Looking for this virtual identity node in the splay tree {
    //never accessed before
    Add it to the splay tree at the most recently
        access position;
    Report an infinite reuse distance;
} else {
    Splay the node to root;
    Report the number of nodes of the sub-tree rooted
        by its right child as the reuse distance;
    Move it to the most recently accessed position;
}
```

Figure 5: The algorithm for the precise reuse distance analyzer

4. EVALUATION WITH SPECJVM2008

In this section, we apply ViRDA to a subset of SPECjvm2008 benchmarks selected based on L2 cache misses per thousand instructions (MPKI).

4.1 Experimental environment

We collected L2 cache MPKI for the 21 non-startup workloads using VTune 9.0 [3] on a 32-bit Windows XP Professional desktop equipped with a 2.83GHz Intel Core 2 Quad Q9550 CPU and 4G DDR2 400MHz memory. The deployed JVM is Apache Harmony [2] with option -Xem:server, -Xms256m and -Xmx256m. For *scimark.fft.large* and *scimark.lu.large*, the options about heap size are changed to -Xms512m and -Xmx512m to prevent OutOfMemory exception. From Table 2, we can see the 4 *scimark* benchmarks [5] with large inputs have the most bandwidth demands followed by *derby*, *compiler.compiler*, *compiler.sunflow* and *serial*.

Workload	L2 cache MPKI
scimark.fft.large	42.16
scimark.lu.large	19.18
scimark.sparse.large	18.64
scimark.sor.large	7.92
derby	4.65
compiler.compiler	2.06
compiler.sunflow	1.69
serial	1.23

Table 2: The top 8 workloads in SPECjvm2008 with the highest L2 cache MPKI

To discover inherent locality profiles, we use ViRDA to analyze the workloads in a single Java thread setting.[2] The workloads are compiled with a basic JIT using -Xem:jet since our focus is on workloads themselves instead of the deployed JVM. The data block size is 64 bytes as a popular configuration. For *scimark* workloads, the precise reuse distance analyzer with a splay tree is used to return precise results even for very short distances for pure temporal locality unless indicated explicitly. For other workloads, the approximate analyzer with a scale tree is used to measure block temporal locality. The precision of approximation is set to 99.9%.

[2]The default setting for SPECjvm2008 workloads is that the number of running threads equals to the number of available cores.

4.2 Slow-down Factor and Component-based Prediction

The main challenge of ViRDA is the significant slowdown. (More details in Section 4.3.7.) But we could use small inputs to predict locality signatures for large inputs where there are inherent locality patterns in an application. This *component-based prediction* is done manually, where a *component* is a group of adjacent bins with non-trivial percentage of total accesses forming a peak in a reuse signature. By changing inputs such as doubling array sizes, we may see some changing patterns for the components (size and resident bin) in the reuse signatures. While not all programs are predictable, the captured patterns can be used to do prediction for some programs, especially the scientific computing applications, as presented by Ding and Zhong [11]. The predictions are verified by running ViRDA with the predicted input sets. We crosscheck the results by examining the source code to ensure the correctness of the locality profile.

4.3 Reuse distance study for workloads

4.3.1 Scimark.fft.large

This workload is a fast Fourier transformation implementation in Java that works on a one-dimensional double array. The reuse signatures are shown in Figure 6. Three small array sizes are tried to infer the changing pattern. For the small inputs, we can see that the first two components are fixed with different array sizes. The third component floats right one bin consistently when the array size is doubled. The predictions of reuse distance signature for DEFAULT and LARGE inputs are drawn in dashed lines. For LARGE input, the third component is in bin 25, which means that those accesses cannot fit a fully associative cache smaller than 16MB. To check the prediction, we use the fast approximation algorithm to verify. Because the approximate algorithm might not be accurate for short distances, we compare only long distances no less than 1KB with the metric

$$Error = \frac{\sum_{i=11} |bin\_predict(i) - bin\_actual(i)|}{2 * min(\sum_{i=11} bin\_predict(i), \sum_{i=11} bin\_actual(i))},$$

where $bin\_predict(i)$ and $bin\_actual(i)$ are the sizes for bin i from prediction and in practice respectively. The prediction Error for LARGE is 6.0%.

The hot method information shows that about 91.7% of long-distance reuses of the third component happen in method *transform_internal()*. Further investigation finds that most of those data accesses happen in a 3-level loop nest in that method. The loop nest can be simplified into the code in Figure 7(a). The *step* used is doubled from 2 exponentially in the outmost loop and the accesses of *data[i+j]* are the cause of long reuse distances when the step size is very large. Fortunately there is a loop interchange opportunity for this loop nest.The transformed loop nest is in Figure 7(b). Spatial locality is improved by the optimization depicted by the block temporal reuse signatures in Figure 8—some long-distance reuses are reduced to bin 7 after the optimization.

We tested the original and optimized code to crosscheck ViRDA results on a real machine with Intel Q9550, where a single thread can use up to 6MB of L2 cache. Figure 9(a) shows a significant distinction of MPKI starts from 8MB

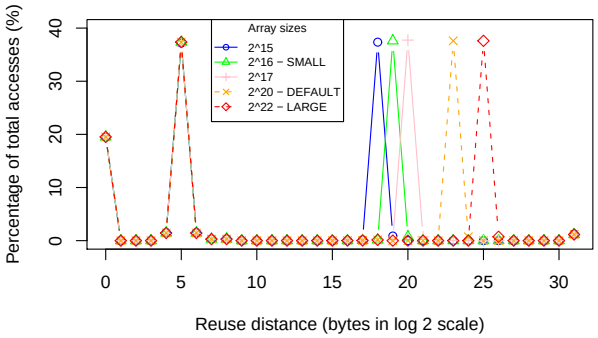


Figure 6: Actual and predicted pure temporal reuse signatures for *scimark.fft*

```
for(int step=2;step<N;step*=2) {
   for(int j=0;j<step;j++) {
      double sum = 0;
      for(int i=0;i<dataSize;i+=step) {
         sum += data[i+j];
         data[i+j] += sum;
      }
   }
}
```

(a) Original code

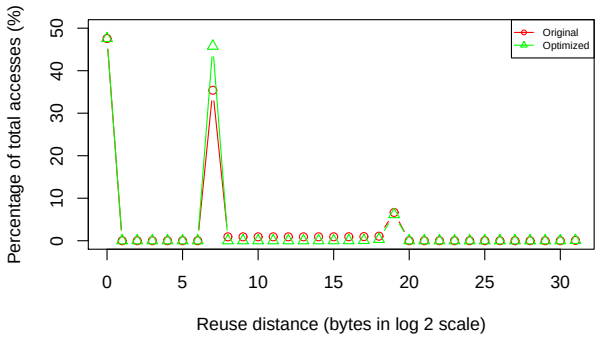
```
for(int step=2;step<N;step*=2) {
   double sum_array[] = new double[step];
   for(int i=0;i<dataSize;i+=step) {
      for(int j=0;j<step;j++) {
         sum_array[j] += data[i+j];
         data[i+j] += sum_array[j];
      }
   }
}
```

(b) Optimized code

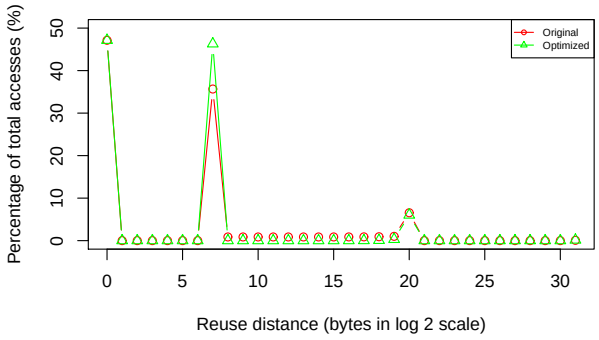
Figure 7: Improving the locality of *scimark.fft* by loop interchange

(23 in X-axis) for the array size. This is consistent with the hardware. When the array is smaller than 6MB, it can be contained in L2 cache and there is almost no cache miss. When the array size continues to grow, L2 cache misses increase sharply with the original loop, but the increase is much slower with the optimized loop. In Figure 9(b), we can see the performance is improved correspondingly.

The block temporal reuse signatures in Figure 8 and Figure 10 explain the results in Figure 9. The reuse signatures for the original code have about 2% accesses for each bin in the middle range (bin 8 to bin 18 in Figure 10(a) and bin 8 to bin 19 in Figure 10(b)). When the array size is doubled, the rightmost component (10% in size) at bin 20 moves one bin right (from bin 19 in Figure 8(a) to bin 20 in Figure 8(b)) and the leftmost two components stay where they are. Each middle bin still carries about 2% of total access but the number of these bins is increased by 1. If the array size is doubled repeatedly, the rightmost component and some of the middle bins on the right will eventually result in cache misses. At that time each doubling causes about 2% of total accesses to become cache misses, which matches



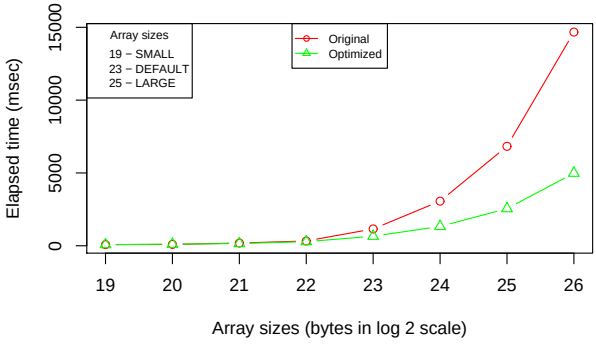
(a) Array length=65536



(b) Array length=131072

Figure 8: Block temporal reuse signatures showing the effect of loop interchange in *scimark.fft*

(a) L2 cache MPKI improvement



(b) Performance improvement

Figure 9: Improvement due to loop interchange in *scimark.fft*

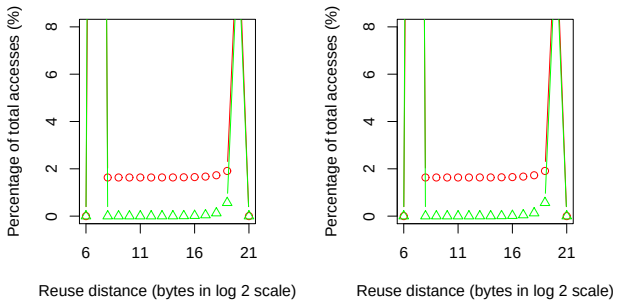
to linear MPKI changes for the original code in Figure 9(a). After loop interchange, the bins in the middle range are almost empty in Figure 10 and the corresponding MPKI curve has a much gentler slope. The rightmost component is the only major contributor of cache misses. The differences in reuse signatures unveil a significant improvement on data locality.

4.3.2 *Scimark.lu.large*

This workload is a Java implementation of LU matrix factorization. It works on a two-dimensional double array. The reuse signatures for different small array sizes are in Figure 11. The first component is fixed but the second component moves to the right by 1 bin and the third component moves to the right by 2 bins when the array size is doubled. The predicted reuse signatures for DEFAULT and almost LARGE (LARGE array size=2048) inputs are the dashed lines. The prediction Error for LARGE is 1.7%. The third component for almost LARGE cannot be contained in a fully associative cache smaller than 16MB. A large number of cache misses are expected to occur at run time. However, due to hardware prefetching, the real performance is better than the predicted because the accesses demonstrate a streaming pattern and the L2 cache MPKI is not as high as that of *scimark.fft.large*. The study of components floating-right with SMALL inputs shows that about 97.7% of the second component and 94.4% of the third component come from the method called *factor()*, in which a 3-level loop nest resides.

4.3.3 *Scimark.sparse.large*

This workload is about sparse matrix multiplication that works on several one-dimensional double or integer arrays. Because the provided configurations for SMALL and DEFAULT are not proportional to the one for LARGE, we do not conduct tests using the SMALL and DEFAULT data sets. Instead, we use other small array sizes proportional to LARGE to infer the changing patterns shown in Figure 12. The first component is fixed but the second component moves to the right by 1 bin when the array size doubled. The reuse signature for LARGE input set is predicted in dashed lines with Error 1.7%. The second component in LARGE could not be contained in a cache smaller than 32MB. The second component mostly comes from the method *matmult()*, the hottest spot of the program with a 3-level loop nest.



(a) Array length=65536 (b) Array length=131072

Figure 10: Local views for middle bins in Figure 8

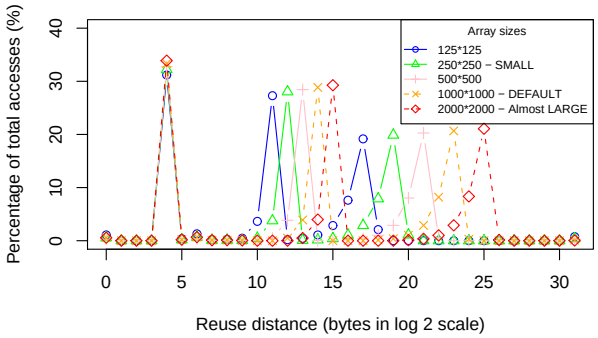


Figure 11: Actual and predicted pure temporal reuse signatures for *scimark.lu*

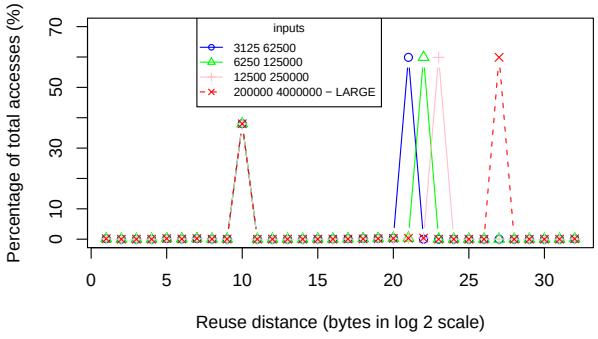


Figure 12: Actual and predicted pure temporal reuse signatures for *scimark.sparse*

4.3.4 Scimark.sor.large

This workload is a Jacobi successive over-relaxation implementation in Java that works on a two-dimensional double array. Reuse signatures for different small array sizes are in Figure 13. When array size is doubled, the first two components are fixed but the third component moves to the right by 1 bin and the fourth component moves to the right by 2 bins. The reuse signatures for DEFAULT and almost LARGE (LARGE array size=2048*2048) inputs are predicted in dashed lines. The prediction Error for LARGE is 3.0%. The fourth component could not be contained in a cache smaller than 16MB. The third and fourth components with SMALL input both come from *execute()* with a 3-level loop nest.

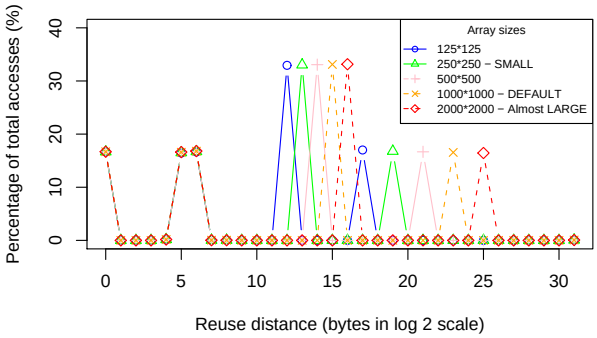


Figure 13: Actual and predicted pure temporal reuse signatures for *scimark.sor*

Let's take a look at why some components are fixed but some move to the right 1 or 2 bins when the array size is doubled. In Figure 14, the fixed components come from short constant reuse distances, e.g. the ones between *A[i][j]* at iteration (X, I, J) and *A[i][j-1]* at iteration $(X, I, J+1)$. The floating-right components come from input-dependent reuse distances, e.g. the ones between *A[i][j-1]* of iteration $(X, I, J+1)$ and *A[i-1][j]* of iteration $(X, I+1, J)$ move to the right by 1 bin and the ones between *A[i-1][j]* of iteration $(X, I+1, J)$ and *A[i+1][j]* of iteration $(X+1, I-1, J)$ move to the right by 2 bins.

```
for (int x=0;x<repeatNum;x++)
    for (int i=1;i<N-1;i++)
        for (int j=1;j<N-1;j++)
            A[i][j] = (A[i+1][j]+A[i-1][j]
                      +A[i][j+1]+A[i][j-1])/4
```

Figure 14: An example illustrating locality components with fixed or changing reuse distances

4.3.5 Derby

This workload is an open source relational database [1]. When running with a single thread, setting it up consumes most of time because only 1 transaction is processed. This is very different from real scenarios. To make results more meaningful, the setting up of the database is skipped and the number of transactions is increased to 10 to focus on transaction processing. This workload is analyzed using the approximate algorithm based on a scale tree.

In Figure 15, the non-trivial long-reuse component at bin 23-28 accounts for more than 4.8% of total accesses. Table 3 shows that the top 5 hottest methods for this component are all constructors for BigDecimal and BigInteger. They

together account for about 65% of this component. Theoretically most accesses in a constructor have infinite distances because the newly created instance is never touched before. The reason might be the duplicated hash codes, which prevents our mechanism from differentiating between two different objects. It is possible that two distinctive objects have the same hash code. That is, the component at bins 23-28 should really be merged to the last bin, since they are caused by constructors. ViRDA relies on the assumption that object hash codes are unique. Although the VM strives to make hash codes different for distinctive objects, it can wrap around if the number of objects is too large.

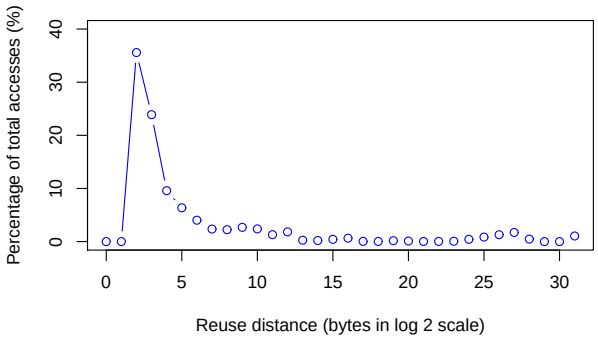


Figure 15: Block temporal reuse signatures for *derby* (transaction processing only)

Method	Method signature	Class	Contribution percentage
<init>	(JI)V	Ljava/math/BigDecimal	34.1%
<init>	(II[I)V	Ljava/math/BigInteger	14.7%
<init>	([CII)V	Ljava/math/BigDecimal	6.7%
<init>	(IJ)V	Ljava/math/BigInteger	5.3%
<init>	(Ljava/math/BigInteger;I)V	Ljava/math/BigDecimal	4.1%

Table 3: Top 5 hot methods for the locality component at bins 23-28 in *Derby*

4.3.6 Compiler.compiler and compiler.sunflow

These two workloads are both about *javac*. The difference is the input sets, which are the source files of *javac* itself and the ones of another workload *sunflow*. Results confirm that they have very similar data locality. Hence only results of *compiler.compiler* are presented here.

Javac compiles all source files of itself in *compiler.compiler* and there is no connection between any two compilations. BoundKind.java is the smallest file and Lower.java is the largest one, which are selected for our analysis. The reuse signature for this whole workload (running with all source files) should be with a shape between the two investigated signatures.

Figure 16 shows that the reuse distance of the component from bin 11 to bin 17 is lengthened with a larger source file. But the effect is much less than the cases for the 4 *scimark* workloads. Table 4 and Table 5 list the details of this component. In Table 4, the column for method signature is omitted because of no overloading. The code and data distribution for this component is flat, which is very different from *scimark* workloads.

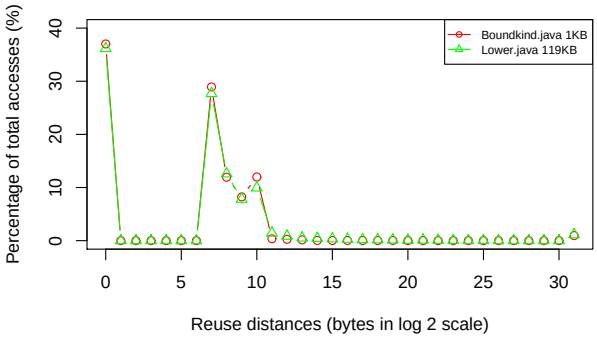


Figure 16: Block temporal reuse signatures for *compiler.compiler*

Method	Class	Contribution percentage
lookup	Lcom/sun/tools/javac/code/Scope;	7.6%
nonEmpty	Lcom/sun/tools/javac/util/List;	6.4%
lookup	Lcom/sun/tools/javac/code/Scope$ImportScope;	5.4%
next	Lcom/sun/tools/javac/code/Scope$ImportScope$ImportEntry;	3.4%

Table 4: Top 4 hot methods for the component at bins 11-17 with Lower.java for *compiler.compiler*

Class	Field	Contribution percentage
Lcom/sun/tools/javac/code/Symbol;	kind	14.0%
Lcom/sun/tools/javac/util/List;	tail	10.0%
Lcom/sun/tools/javac/code/Type;	tag	8.3%
Lcom/sun/tools/javac/code/Scope$Entry;	sym	8.0%

Table 5: Top 4 hot class & fields for the component at bins 11-17 with Lower.java for *compiler.compiler*

The hottest method, *lookup()*, is related to an *Entry* hash table. The lookup operation checks field *sym* of *Entry* instances. Since the number of entries of the hash table is fixed at 16, it is highly possible that bad hashing happens here—too many elements are mapped to the same table entry. In the source code, we found a comment saying "the adaptive expanding turned off due to some bug". We tried turning it on anyway but didn't encounter any errors. The contribution from this method is reduced to 4.4%. From the data view, the contribution from instance field *sym* of class *Entry* is reduced to 2.5%. And the actual performance for the whole workload is improved by about 2%, although the improvement may be unstable.

4.3.7 Serial

This workload performs 16 kinds of serializations that are studied separately using the approximate algorithm based

on scale tree. Some statistics for each sub-workload are in Table 6. It shows ViRDA with about 250x slowdown for all these sub-workloads. In results, only 3 of them (*Array*, *ArrayList* and *Payload*) have non-trivial part ($\geq 1\%$) of reuse distances in bins greater than 10, which are plotted in Figure 17. The three sub-workloads are with longer access trace and/or larger data set. The component for *Array* moves to the right by 1 bin if the array size is doubled. It is consistent with the previous findings in *scimark* workloads.

Sub-workload	Access trace length	Data set size (Byte)	Prof/exe elapsed time
Array	1.1e+9	4.1e+5	39m46s/8422ms
ArrayList	5.9e+8	6.8e+6	26m4s/5157ms
ByteArray	1.0e+8	3.9e+5	6m37s/1031ms
ClassReference	2.5e+7	3.9e+5	4m2s/937ms
ClassWithSQL DateOnly	2.5e+7	3.9e+5	4m5s/969ms
DomainObject	6.5e+7	4.0e+5	5m21s/1234ms
Exception Reference	3.9e+7	4.0e+5	4m26s/1046ms
Externalizable	2.5e+7	3.9e+5	4m1s/922ms
HugeData	2.5e+7	4.7e+5	4m21s/1109ms
Parent	1.2e+8	4.3e+5	7m37s/1515ms
Payload	1.1e+8	1.7e+7	9m28s/5516ms
Proxy	3.3e+7	3.9e+5	6m/3906ms
ReadResolve	2.7e+7	3.9e+5	4m6s/922ms
Simple	2.5e+7	3.9e+5	4m13s/1063ms
WithBigDecimal	4.9e+7	3.9e+5	5m5s/1141ms
WithFinalField	2.5e+7	3.9e+5	4m4s/938ms

Table 6: The access length, data set size and elapsed time for each sub-workload of *serial*

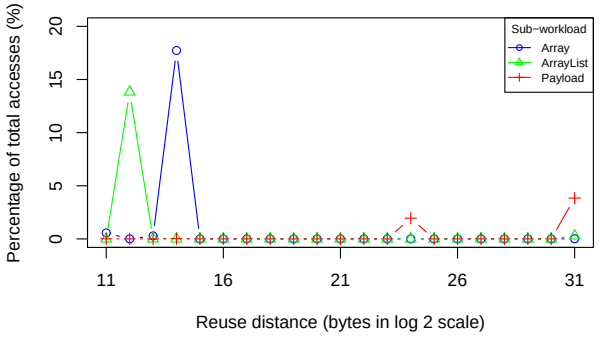


Figure 17: Block temporal reuse signatures for *Array*, *ArrayList* and *Payload* in *serial*

The details for the non-infinite components for *Array* and *Payload* are listed in Table 7, Table 8 and Table 9. (The results for *ArrayList* are omitted since they are very similar to those of *Array*.) The distributions are all very concentrated. Besides the listed hot classes and fields, character arrays accounted for about 40% or 100% for *Array* and *Payload* respectively.

Method	Method signature	Class	Contribution percentage
equals	(Ljava/lang/ Object;)Z	Ljava/security/ CodeSource	59.1%
regionMatches	(ILjava/lang/ String;II)Z	Ljava/lang/String;	39.4%

Table 7: Top 2 hot methods for the component at bin 14 for *Array* of *serial*

Class	Field	Contribution percentage
Lcom/sun/tools/ javac/code/Symbol;	kind	14.0%
Lcom/sun/tools/javac/util/List;	tail	10.0%
Lcom/sun/tools/javac/code/Type;	tag	8.3%
Lcom/sun/tools/javac/ code/Scope$Entry;	sym	8.0%

Table 8: Top 2 hot classes & fields for the component at bin 14 for *Array* of *serial*

Method	Method signature	Class	Contribution percentage
toString	(I)Ljava/lang/ String;	Ljava/lang/Integer;	49.7%
regionMatches	(ILjava/lang/ String;II)Z	Ljava/lang/String;	39.4%

Table 9: Top 2 hot methods for the component at bin 14 about *Payload* of *serial*

5. LIMITATIONS AND FUTURE WORK

The main concern for ViRDA is its time cost. Time distance [21, 20] or sampling techniques [15, 25] could alleviate this problem by reducing the measurement time by an order of magnitude. We may make analysis even faster by combining these methods [8].

Another concern is hash code conflicts—the same identity hash code for different objects. In JVM, the hash code of an object usually equals to the memory address when it is first used. If too many objects are allocated, conflicts may happen as the case in *Derby*. We could improve the hash code generation mechanism by replacing the high-end bits of hash code with the number of garbage collections that have happened since only low-end bits of memory address are different in nursery space.

The connection between ViRDA and traditional reuse distance analysis is also interesting. Some investigations combining the two kinds of different results on reuse distance need to be done.

6. SUMMARY

Reuse distance analysis has been proven effective in analyzing program data locality. It is especially useful to identify program code that causes poor data locality. In prior efforts, memory addresses were used while working with programs written in languages like Fortran or C/C++. In managed run time environments, however, simply using memory addresses could result in inaccurate analysis due to the interference of runtime support such as data movement. We propose ViRDA that identifies program data using symbolic information in reuse distance analysis for runtime applications. ViRDA only profiles accesses directly issued by applications, bypassing runtime internal data accesses and the data relocation effect of GC. Furthermore, ViRDA is built

on top of standard runtime profiling interface, which makes it independent of underlying runtime implementations.

ViRDA is used in locality analysis for 8 workloads in SPECjvm2008 that have the worst locality. Due to the locality nature for applications, we can use small input sets in practical profiling and project collected reuse signatures for large input sets with small errors. We expect the SPECjvm2008 locality profiles by ViRDA could help the community to better understand the workloads behavior and optimize Java virtual machine systems for improved performance.

Acknowledgment

We wish to thank Chen Ding in University of Rochester for his insightful technical suggestions and kindly help on paper revision, and the anonymous reviewers for their valuable comments.

References

[1] Apache Derby. `http://db.apache.org/derby`.

[2] The Apache Software Foundation, Apache Harmony. `http://harmony.apache.org`.

[3] Intel VTune. `http://software.intel.com/en-us/intel-vtune`.

[4] JVM Tool Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti`.

[5] SciMark 2.0. `http://math.nist.gov/scimark2`.

[6] Standard Performance Evaluation Corporation, SPECjvm2008. `http://www.spec.org/jvm2008`.

[7] B. T. Bennet and V. J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[8] K. Beyls and E. H. D'Hollander. Discovery of locality-improving refractorings by reuse path analysis. In *HPCC*, pages 220–229, 2006.

[9] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, pages 229–241, 1999.

[10] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[11] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257, 2003.

[12] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application. In *IEEE PACT*, pages 27–37, 2005.

[13] X. Gu, I. Christopher, T. Bai, C. Zhang, and C. Ding. A component model of spatial locality. In *ISMM*, pages 99–108, 2009.

[14] M. S. Lam and M. E. Wolf. A data locality optimizing algorithm (with retrospective). In *Best of PLDI*, pages 442–459, 1991.

[15] K.-H. Li. Reservoir-sampling algorithms of time complexity O(n(1 + log(N/n))). *ACM Transactions on Mathematical Software (TOMS)*, 20(4):481–493, December 1994.

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

[17] S. A. McKee. Reflections on the memory wall. In *Conf. Computing Frontiers*, page 162, 2004.

[18] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. *Technical Report LBL-12370, Lawrence Berkeley Laboratory*, 1981.

[19] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *ICS*, pages 131–140, 2005.

[20] X. Shen and J. Shaw. Scalable implementation of efficient locality approximation. In *LCPC*, pages 202–216, 2008.

[21] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL*, pages 55–61, 2007.

[22] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, pages 165–176, 2004.

[23] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. Specjvm2008 performance characterization. In *SPEC Benchmark Workshop*, pages 17–35, 2009.

[24] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[25] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *ISMM*, pages 91–100, 2008.

[26] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, pages 255–266, 2004.