PARDA: A fast parallel reuse distance analysis algorithm

Conference Paper · May 2012

DOI: 10.1109/IPDPS.2012.117

4 authors, including:

Qingpeng Niu
Facebook
11 PUBLICATIONS **66** CITATIONS
SEE PROFILE

James Dinan
Intel Corporation, Hudson, MA
69 PUBLICATIONS **1,182** CITATIONS
SEE PROFILE

Ponnuswamy Sadayappan
The Ohio State University
410 PUBLICATIONS **9,602** CITATIONS
SEE PROFILE

PARDA: A Fast Parallel Reuse Distance Analysis Algorithm

Qingpeng Niu
The Ohio State University
niuq@cse.ohio-state.edu

James Dinan
Argonne National Laboratory
dinan@anl.gov

Qingda Lu
Intel Corporation
qingda.lu@intel.com

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

***Abstract*—Reuse distance is a well established approach to characterizing data cache locality based on the stack histogram model. This analysis so far has been restricted to offline use due to the high cost, often several orders of magnitude larger than the execution time of the analyzed code. This paper presents the first parallel algorithm to compute accurate reuse distances by analysis of memory address traces. The algorithm uses a tunable parameter that enables faster analysis when the maximum needed reuse distance is limited by a cache size upper bound. Experimental evaluation using the SPEC CPU 2006 benchmark suite shows that, using 64 processors and a cache bound of 8 MB, it is possible to perform reuse distance analysis with full accuracy within a factor of 13 to 50 times the original execution times of the benchmarks.**

***Keywords*-Data Locality; Reuse Distance; LRU Stack Distance; Caching; Performance Analysis**

I. Introduction

Caching is a fundamental performance optimization technique in computer science. A cache is a small, fast memory that transparently stores frequently referenced data so that future requests for that data can be served faster. If requested data is already in the cache (i.e., a cache hit), such a request can be served much faster from cache than from its original storage location (upon a cache miss). For over two decades, the performance gap between the processor and DRAM has been increasing at an exponential rate. This "memory wall" problem has been alleviated by a multi-level processor cache design. In addition, an increasing number of processing cores share cache at different levels in the system. Thus, cache performance plays a critical and evolving role in determining the speed, cost, and energy usage of most modern computer systems for many applications.

The effectiveness of caching depends on data locality and the pattern of data reuse. Reuse distance (also called LRU stack distance) is a widely used metric that models data locality [12], [4]. Defined as the number of distinct memory references between two successive references to the same location, reuse distance provides a quantification of the locality present in a data reference trace. Reuse distance has two important advantages over other locality models. (1) Most cache designs use the LRU (Least Recently Used) replacement policy or its variants. Therefore with a cache of size N, all references with reuse distance $d \leq N$ would be very accurately modeled as hits and all others, misses. (2) Reuse distance measures the volume of the intervening data between two accesses and is always bounded by the size of physical data, while other metrics such as time distance can be unbounded.

Since its introduction in 1970 by Mattson et al. [12], reuse distance has found numerous applications in performance analysis and optimization, such as cache miss rate prediction [20], [11], [8], program phase detection [16], data layout optimization [21], virtual memory management [3] and I/O performance optimization [7]. A major challenge facing these reuse-distance-based applications has been the extremely high cost of the analysis, severely limiting online use of the analysis for dynamic performance optimization. While significant efforts have been made to accelerate the reuse distance computation's search structures [13], [18], [1], the computational complexity is still not satisfactory and therefore many researchers have resorted to trading off accuracy for speed [4], [19], [15].

In this paper, we present *Parda*, a fast parallel algorithm to compute accurate reuse distances by analysis of reference traces. The algorithm uses a tunable parameter that enables faster analysis when the maximum needed reuse distances is limited by a cache size upper bound. To our knowledge, this paper presents the first parallel algorithm that performs precise, single-trace reuse distance analysis. Experimental evaluation using the SPEC CPU 2006 benchmark suite shows that using 64 processors it is possible to bring down the overhead of reuse distance analysis to within a factor of 13 to 53 times the original execution time of the benchmarks; this contrast to slowdown factors of several hundreds to thousands from previous algorithms. In this work, we have chosen a popular tree-based algorithm as our core analysis engine, but Parda is compatible with many other sequential data reference analysis algorithms, enabling a broad range of uses. Our algorithm can form the basis for efficient reuse distance analysis using increasingly available multi-processor systems. We also expect that with the adoption of many-core processors and accelerators into commodity systems, parallel reuse distance analysis will benefit a larger class of systems and applications, in particular those that rely on online analysis, such as cache sharing and partitioning [14], [9].

The rest of the paper is organized as follows. We first provide background information about reuse distance analysis in Section II. We then describe the basic sequential algorithm

Time	0	1	2	3	4	5	6	7	8	9
Data Ref.	d	a	c	b	c	c	g	e	f	a

Table I
EXAMPLE DATA REFERENCE TRACE

and its popular tree-based implementation in Section III. We present our general parallel algorithm *Parda* and prove its correctness in Section IV. We then develop the tunable algorithm with cache bounds in Section V. We then discuss our implementation of Parda based on message passing (MPI) and binary translation, and evaluate the effectiveness of our approach in Section VI on a cluster. We discuss related work in Section VII and finally present our conclusions.

II. BACKGROUND: REUSE DISTANCE

Reuse distance (also frequently referred to as LRU stack distance) is a quantification of the locality present in a data reference trace. The reuse distance of a memory access is the number of access to distinct addresses made since the last reference to the same data. Assuming a fully associative cache of size N, all references with distance $d \leq N$ would be hits and all others, misses.

An example data reference trace is shown in Table I. This example trace is of length $N = 10$ and contains references to $M = 7$ distinct data elements. We define the data reference sequence Ψ_s^e to be a time ordered sequence of data references beginning at time s and ending at time e.

$$\Psi_s^e = \begin{cases} \langle \Psi[s], \Psi[s+1], \ldots, \Psi[e] \rangle & s \leq e \\ \emptyset, & s > e \end{cases} \quad (1)$$

For the example data reference trace Ψ given in Table I, $\Psi_1^5 = \langle \Psi[1], \Psi[2], \ldots, \Psi[5] \rangle = \langle a, c, b, c, c \rangle$. We define the cardinality of such a sequence,

$$|\Psi| = |\{i : i \in \Psi\}| \quad (2)$$

to be the number of unique elements in Ψ. For example, $|\Psi_1^5| = 3$. For a given sequence, we define the set of indexes where element z is referenced as,

$$R_z(\Psi_s^e) = \{i : s \leq i \leq e \wedge \Psi[i] = z\} \quad (3)$$

For example, in the example trace $R_c(\Psi_1^5) = \{2, 4, 5\}$. Using this construct, we define the maximal index for data element z in a given sequence as,

$$Max_z(\Psi_s^e) = \begin{cases} \max R_z(\Psi_s^e), & R_z(\Psi_s^e) \neq \emptyset \\ \emptyset, & \text{otherwise} \end{cases} \quad (4)$$

Using the above, $Max_c(\Psi_1^5) = \max\{2, 4, 5\} = 5$. Thus, the reuse distance for a reference to element z at time i can be expressed as,

input : Ψ_0^{N-1}: An N-entry data reference sequence.
output: $hist$: Array of integers containing the histogram of reuse distances for all references in Ψ.
let:
T be a binary tree
H be a hash table
$hist[M]$ be an array of integers
$T \leftarrow \emptyset$
$H \leftarrow \emptyset$
$hist[\ldots] \leftarrow 0$
for $t \leftarrow 0$ **to** $N-1$ **do**
  $z \leftarrow \Psi_0^{N-1}[t]$
  $d \leftarrow \infty$
  if $H(z) \neq \emptyset$ **then**
    $d \leftarrow$ distance$(T, H(z))$
    $T \leftarrow$ delete$(T, H(z))$
  end
  $hist[d] \leftarrow hist[d] + 1$
  $T \leftarrow$ insert(T, z, t)
  $H(z) \leftarrow t$
end

Algorithm 1: Tree-based sequential reuse distance analysis algorithm.

$$D_z(i) = \begin{cases} |\Psi_{Max_z(\Psi_0^{i-1})+1}^{i-1}|, & Max_z(\Psi_0^{i-1}) \neq \emptyset \\ \infty, & \text{otherwise} \end{cases} \quad (5)$$

For the example given above, $D_c(4) = |\Psi_3^3| = 1$.

III. SEQUENTIAL REUSE DISTANCE ANALYSIS

Reuse distance analysis can be extremely useful, but also costly. Consequently, many approaches and optimizations have been proposed in the literature, ranging from the full analysis [2], [13] to probabilistic approaches that trade accuracy for performance [4], [19]. To our knowledge, our work is the first attempt at parallelizing reuse distance analysis to leverage multiple cores or nodes in a parallel system to accelerate the analysis.

Our approach to parallel reuse distance analysis is compatible with many existing sequential data reference analysis algorithms. The development in this paper utilizes the efficient sequential reuse distance analysis algorithm developed by Olken et al. [13].

A. Naïve Algorithm

A naive approach to reuse distance analysis uses a stack data structure to maintain an ordered list of data references. This stack simulates the behavior of an infinite sized, fully associative cache. Initially the stack is empty and data references are pushed onto the head in trace order. When pushing a reference, the stack is traversed starting at the

head to determine if there was a previous reference to this data element. If the data reference is found, the old instance is removed and pushed onto the head. The the distance from the head to the position of the old instance is recorded as the stack, or reuse distance for the new reference. If the data reference is not found, this is the first reference to this data element and it is recorded as having a distance of infinity (corresponding to a compulsory cache miss).

For a trace of length N containing M distinct references, N elements must be processed and at each step an $O(M)$ traversal of the stack is performed. Thus, the asymptotic running time of the naive algorithm is $O(N \cdot M)$. The space complexity is $O(M)$ since the list can contain at most one entry per distinct data address in the reference trace.

B. Tree-Based Algorithm

Several tree-based reuse distance analysis algorithms have been developed to reduce the $O(M)$ cost of stack traversal in the naive algorithm to $O(\log M)$. Often a splay tree [17] is used to enhance the efficiency of this structure. These algorithms generally make use of two data structures: a hash table to store the timestamp of the most recent reference to each data element and a tree that can be used to efficiently determine the number of distinct references since the last time a data element was referenced.

Algorithm 1 presents the popular tree-based reuse distance analysis algorithm [13]. In this algorithm, a hash table H maps data references to their most recent access timestamp. A binary tree T is maintained that holds one entry for each data reference seen thus far and is sorted according to timestamp. Subtree sums are maintained at each node for the number of elements in the subtree rooted at the node. Thus the process of counting the distance associated with a data reference is that of traversing the tree from the root to that data reference and accumulating the sum along the way.

As shown in Figure 1, the algorithm maintains a balanced binary tree with at most one entry per data reference. Each node contains the reference, the timestamp of the last access, and the sum of the weights in its subtree. Insertion and deletion are performed using the standard algorithm for balanced binary trees, which ensures that the ordering is preserved and that the tree remains balanced. When performing the distance operation, we find the timestamp t of the last access to the current data element using the hash table and then apply the distance calculation algorithm shown in Algorithm 2.

Figure 1(a) shows the tree state before and after processing reference 'a' at time 9 for the data reference trace in Table I. When processing this reference, the tree-based algorithm first checks whether H contains 'a'. For this reference, 'a' is found to have a previous timestamp of 1. Starting at the root of the tree, the algorithm adds one plus the weight of the right subtree and then moves to the root's

```
input : T, a data reference tree
        t, desired timestamp
output: d, the number of the nodes in T whose
        timestamp larger than t
cur ← T.root
d ← 0
while true do
    if t.ts > cur.ts then
        cur ← cur.right_child
    else if t.ts < cur.ts then
        d ← d + 1
        if cur.right_child ≠ ∅ then
            d ← d + cur.right_child.weight
        end
        cur ← cur.left_child
    else
        d ← d + cur.right_child.weight
        return d
    end
end
```

Algorithm 2: Tree distance algorithm.

left child. This is the node containing 'a', so the search has completed and one is added to the distance for the current node yielding a final distance of $d = 1 + 3 + 1 = 5$.

After distance operation completes, the node containing 'a' is deleted from the tree and added with the new timestamp of one. The resulting tree is shown in Figure 1(b).

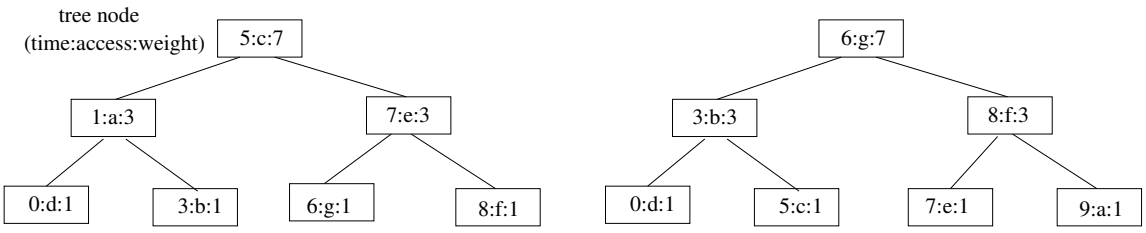
In this algorithm, we perform N analysis steps, one for each entry in the reference trace. Each step consists of a hash table lookup and replacement which are $O(1)$ in cost and up to three tree traversals which are each $O(\log M)$ in cost. Thus, the time complexity is $O(N \cdot \log M)$. In terms of space, the tree, hash table, and histogram all require space proportional to M, resulting in a space complexity of $O(M)$.

IV. Parallel Reuse Distance Analysis

At first glance, reuse distance analysis appears to exhibit strong data dependence on previous references, inhibiting parallelism. However, an inherent property of reuse distance analysis is that the chain of dependence for any given data item only reaches back as far as its last reference. In other words,

Property 4.1: A reference to any given data element at time t whose last reference occurred at t' is independent of all references that occurred before t'.

Using this insight, we have taken a chunking approach to parallelizing reuse distance analysis that aims to exploit this independence to achieve higher performance. In this approach, we divide the data reference trace into p chunks and assign each chunk to one of p processors. We then apply a modified version of the sequential algorithm described in Section III to each chunk. All non-infinite references



(a) Tree before processing reference 'a' time 9.

(b) Tree after processing reference 'a' at time 9.

Figure 1. Tree state before and after processing the reference at time 9 for the data reference trace in Table I.

input : Ψ_0^{N-1}: An N-entry data reference sequence.
output: $hist$: Array of integers containing the histogram of reuse distances for all references in Ψ.
Let:
np be the number of processors
$p \in \{0, \ldots, np-1\}$ be the rank of this proc
T, H be the tree and hashtable
$hist[M]$ be an M-length array of integers
i be an integer iteration counter

$C \leftarrow N/np$
$S \leftarrow \Psi_{C*p}^{C*(p+1)-1}$
$T \leftarrow \emptyset$
$H \leftarrow \emptyset$
$hist[\ldots] \leftarrow 0$
$i \leftarrow 0$
while $i < np - p$ **do**
  $(T, H, hist, local\_infinities) \leftarrow stack\_dist(T, H, hist, S)$
  if $p > 0$ **then**
    $Send(local\_infinities, p-1)$
  else
    $hist[\infty] \leftarrow hist[\infty] + |local\_infinities|$
  end
  if $p < np - 1$ **then**
    $S \leftarrow Recv(p+1)$
  end
  $i \leftarrow i + 1$
end
$hist \leftarrow reduce\_sum(hist)$

Algorithm 3: The *Parda* parallel reuse distance analysis algorithm.

are correct by Property 4.1 since chunks are contiguous and contain all references between t' and t. However, *local infinities*, or infinite distance references found when processing chunk k must be passed downward toward the process working on chunk 0 by appending the references to chunk $k-1$. If these references reach chunk 0 and are still found to be infinities, they will be counted as *global infinities* and tallied as such in the histogram.

An example of this is shown in Table II. In this table we divide the data reference trace into two chunks which are processed in parallel. The local reuse distance is the result arrived at when processing only the local chunk and the global reuse distance is the global distance when considering the full trace. We can see that intra-chunk hits result in correct distances, however some local infinities of the right chunk were not global infinities.

A. The Parda Algorithm

The Parda algorithm is presented in Algorithm 3. In this algorithm, we perform np steps where each process processes its chunk of the trace and then sends its local infinities to its left neighbor. Likewise, each process also receives the local infinities of its right neighbor and processes this reference sequence continuing from its current state.

Processing of the current reference sequence is accomplished by a minimally modified version of the $stack\_dist$ function presented in Algorithm 1. This new algorithm has been modified so that the state of the analysis (the tree, hash table, and histogram) can be passed in as input and returned as output, allowing the analysis to continue from its current state if new trace information is received. In addition, rather than counting the local infinities in the histogram, these references are maintained in a time-ordered queue and returned as the $local\_infinities$ sequence. The $local\_infinities$ sequence contains the ordered list of references that were of infinite distance, given the current state of the analysis and it has several important properties including,

Property 4.2: The local infinities sequence for chunk k contains one entry for each distinct element in chunk k.

As infinities are processed and passed to the left, each process filters out references to data elements that it has already seen and updates its histogram. Thus, processing local infinities can be viewed as compressing the trace down to its unique elements.

Property 4.3: The complete sequence of local infinities produced by process p contains one entry for each distinct element in chunks $p \ldots np-1$.

Once the analysis has completed, a parallel reduction is performed on the histogram array to accumulate the results from all processes.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Data Referenced	**d**	**a**	**c**	**b**	c	c	**g**	**e**	**f**	**a**	f	**b**	**c**
Local Distance	∞	∞	∞	∞	1	0	∞	∞	∞	∞	1	∞	∞
Global Distance	∞	∞	∞	∞	1	0	∞	∞	∞	**5**	1	**5**	**5**

Table II
EXAMPLE TWO PROCESSOR REUSE DISTANCE HISTOGRAM COMPARING LOCAL AND GLOBAL REUSE DISTANCES.

B. Proof of Correctness

In order to demonstrate that the parallel algorithm is correct, we must show that for a data reference sequence Ψ_0^{N-1}, the reuse distances are calculated correctly. Assuming a chunk size of C, a process p is assigned the subsequence $\Psi_{p\cdot C}^{(p+1)\cdot C-1}$, we break this down into three cases:

1. Element $e \in \Psi_{p\cdot C}^{(p+1)\cdot C-1}$ is referenced at time t and its most recent previous reference time $p\cdot C \leq t' \leq (p+1)\cdot C-1$.

This element's previous reference occurs within the same chunk and is guaranteed to be counted correctly by Property 4.1.

2. Element $e \in \Psi_{p\cdot C}^{(p+1)\cdot C-1}$ is referenced at time t and it is the first reference to this element (i.e. a global infinity).

Because this is the first reference to this item, $e \notin \Psi_0^{p\cdot C-1}$, and none of the processors with rank lower than p have a reference to this item. Thus, the reference to e at time t will be also be a local infinity on all lower ranked processes and will pass through each process' local infinities list. When process 0 receives this element, it will also not be found and will be counted as a global infinity.

3. Element $e \in \Psi_{p\cdot C}^{(p+1)\cdot C-1}$ is referenced at time t and its most recent previous reference time $0 \leq t' \leq p \cdot C - 1$.

This element will be passed to processor $p-1$ as a local infinity and will ultimately be received by the processor p' that has the previous reference to e at time t' in its tree. By Property 4.3, this local infinities sequence sent by processor $p'+1$ will contain all unique references in $\Psi_{p'+1\cdot C}^{t}$. These references will be processed by p' using Algorithm 1 and entered into its tree. When the reference to e is encountered, p''s tree will contain all unique references that occurred between time t' and t and, thus, will count the correct number of unique references between these two times yielding the correct result.

C. Space-Efficient Local Infinity Processing

In the Parda algorithm presented in Algorithm 3, all processes add their neighbor's local infinities to their tree and hash table even if they are forwarded along as local infinities. By Property 4.3, all entries in the aggregate local infinities list for any process are unique, so it is unnecessary to store them in the tree or hash table since they will not occur again in the local infinities list received by this processor. However, their contribution to the reuse distance of any future hits is still needed. Parda can be extended to exploit this property to eliminate duplication across processors, resulting in significant space savings.

In Algorithm 4 we present a local infinities processing function that is used to during the local infinities processing stage, after the initial chunk of the data reference trace has been processed. In this algorithm, each process maintains a counter of the total number of local infinities that were received. When a hit occurs, the distance is calculated according to existing Algorithm 1 and this count is added to offset the time by the number of additional unique references that have occurred but are not accounted for in the tree. In addition, because the hit has occurred due to an incoming local infinity, we know that the reference in our tree is a replica of a tree node in our right neighbor. Because of this and because by Property 4.3 we know that we will not receive this reference again in the local infinities sequence, it can be removed from our tree.

```
input : H, T, hist
        local_infinities, the local inf list
        count, the current local infinities count
output: H, T, hist
        local_infinities', new local inf list
        count, the new local infinities count
local_infinities' ← ∅
for i ← 0 to |local_infinities| − 1 do
    z ← local_infinities[i]
    if H(z) ≠ ∅ then
        d ← distance(T, H(z)) + count
        hist[d] ← hist[d] + 1
        T ← delete(T, H(z))
        H(z) ← ∅
    else
        local_infinities'.append(z)
    end
    count ← count + 1
end
```

Algorithm 4: Space optimized local infinities processing algorithm.

We present an example of the optimized algorithm in Figure 2. This example shows the tree structure, local infinities, and count on each processor for the data reference trace in Table III. The top row shows the state of each process after processing its chunk of the trace. The bottom

row shows the state of processors 0 and 1 after processing their neighbor's local infinities. As infinities are processed, entries are deleted from process 0 and 1's trees. The final state of all processes consists of the trees shown in Figures 2(c), 2(e), and 2(f). From this, we can see that each reference appears at most once on all processes.

By property 4.3, the local infinities list will effectively wipe out all duplicates as it is sent to the left. Thus, when compared with the original algorithm whose final aggregate space usage grew linearly in the number of processors $O(np \cdot M)$, this algorithm's final aggregate space usage is $O(M)$. The time complexity of the parallel algorithm includes a $O(\frac{N}{P} \cdot log\frac{N}{P})$ component for the local processing and a $O(log\frac{N}{P} \cdot M)$ worst-case bound for the merge step. This corresponds to a scenario where the distinct elements forming the local-infinities of the last processor are not present in any of the other processors, thereby requiring processing by each processor in turn before reaching the first processor.

D. Multiple-Phase Algorithm

input : $INPUT$, a streaming input source
C, the chunk size
output: $hist$, array of integers containing the histogram of reuse distances for all references in $INPUT$.
let : T, H be the tree and hash table
repeat
  $count \leftarrow$ read$(INPUT, S)$
  $(T, H, hist) \leftarrow$ parda$(S_{p \cdot C}^{(p+1) \cdot C - 1}, C)$
  $(T, H) =$ reduce_parda_state(T, H)
until $count < np \cdot C$
$hist \leftarrow reduce\_sum(hist)$

Algorithm 5: Phase-based Parda algorithm.

Algorithm 3 assumes that we can divide the entire trace into chunks at the beginning of the analysis. However, in practical applications the length of a streaming trace may be unknown and traces stored for offline analysis can easily contain 100 billion references or more for only several seconds runtime. In Algorithm 5, we present an enhanced algorithm that allows Parda to analyze the entire trace online through S phases. At each phase, we process 1/S of the trace and in this way reuse distance analysis can be conducted in parallel with tracing. In this algorithm, processor with id 0 holds the global state for the timestamp that marks the beginning of the current phase. This preserves the property that process 0 is able to give the authoritative answer for global infinities. Initially the global state is empty, however after each phase is completed, the state must be accumulated onto the process with id 0. At the end of each parda phase, each processor performs Algorithm 6. Algorithm 6 describes the reduction step. All processors other than

input : H, T
output: H,T
Let:
  np be the number of processors
  $p \in \{0, \ldots, np-1\}$ be the rank of this proc

if *p=np-1* **then**
  for $dest \leftarrow np-1$ **to** *0* **do**
    $H_L \leftarrow Recv(dest)$
    foreach $[e,t] \in H_L$ **do**
      insert(T,e,t)
      H(e)=t
    end
  end
  send(H,0)
  send(T,0)
else
  send(H,np-1)
end

Algorithm 6: reduce_parda_state

processor np-1 send the contents of H – the data accesses and their most recent timestamps – to processor np-1. Processor np-1 merges received accesses and timestamps for the current phase and send global state back to processor 0 before the next phase starts. With Algorithm 4, on processor np-1 we do not need duplication check when inserting data into the global hash table and the tree as duplicated accesses have already been deleted. As the merging process begins from processor np-1 in the reverse order of processor ids, while the other processors are busy processing local infinities receiving from right, processor np-1 synchronizes with free processors to merge the global state. This makes multiple-phase algorithm achieves good load balancing.

Further enhancement of this multiple-phase algorithm can be achieved as follows: At the end of each phase, as processor np-1 holds the global state of the trace processed so far, there is no need to communicate it back to processor 0. We can reassign processor ids in the reverse order therefore processor np-1 becomes the processor 0 at next phase.

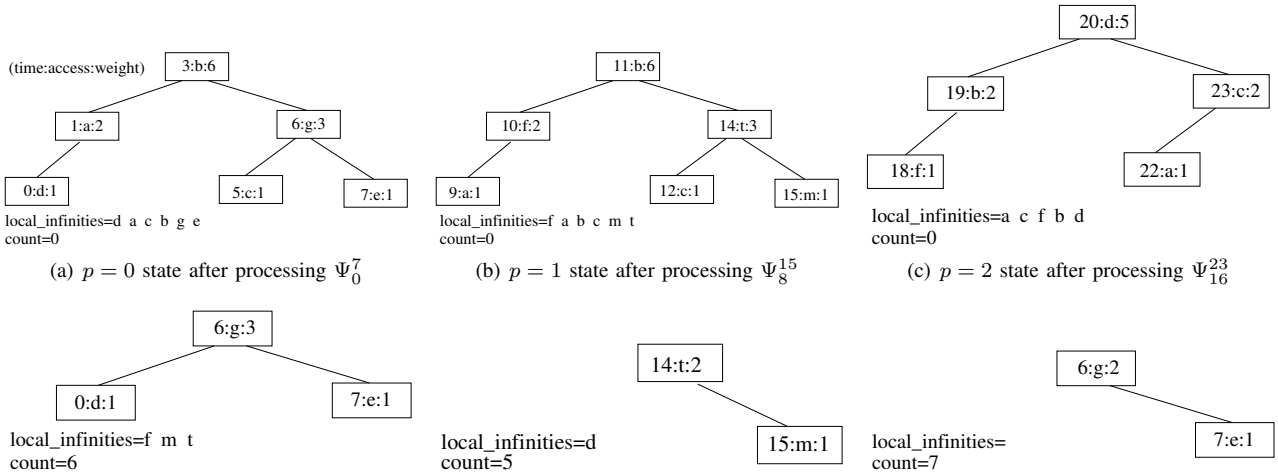
V. Bounded Parallel Reuse Distance Analysis

In practice, any cache, whether it be processor cache, disk, virtual memory, or others, has an upper bound in size beyond which additional analysis information is not useful. We can exploit this upper bound to improve both the time and space complexity from M, the number of distinct elements in the trace, to C, the cache size. In general and especially for large traces, this can result in significant performance improvement.

In order to achieve this functionality, we modify the $stack\_dist$ algorithm used by Parda (Algorithm 3) and replace it with a new bounded cache algorithm presented

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Data Ref.	d	a	c	b	c	c	g	e	f	a	f	b	c	m	t	m	a	c	f	b	d	c	a	c

Table III
EXAMPLE DATA REFERENCE TRACE



(a) $p = 0$ state after processing Ψ_0^7 (b) $p = 1$ state after processing Ψ_8^{15} (c) $p = 2$ state after processing Ψ_{16}^{23}

(d) $p = 0$ state after processing first round of local infinities from $p = 1$. (e) $p = 1$ state after processing local infinities from $p = 2$. (f) $p = 0$ state after processing second (final) round local infinities from $p = 1$.

Figure 2. Tree state for a three processor space-optimized Parda analysis of the trace in Table III (top) after processing the chunk of the data reference trace and (bottom) while applying the optimized local infinities processing algorithm.

in Algorithm 7. The only modification to the main Parda algorithm is the addition of l, the total number of local infinities generated, which is initialized to 0 and passed into the reuse distance algorithm.

In this algorithm, we set an upper bound B on the number of entries in the hash table and tree data structures. In addition, at most B local infinities may be produced to pass to neighbor processes. This is because all misses beyond the first B are capacity misses due to replacement. As such, we can immediately count them locally as infinities without the need for further analysis.

When processing a reference, we first check to see if it exists in the hash table. If so, it is processed normally. If the reference is not located, we must determine which of three possible cases has occurred. When $l < B$, we are permitted to add this reference to the local infinities list to send to our neighbor for processing. Otherwise, we must count this reference as a global infinity and handle the reference locally. If the hash table is full, we must remove something from H and T to make space for the new reference, otherwise it can simply be added.

Replacement is performed by a depth first traversal to the leftmost leaf node. Because the tree is sorted, this node has the oldest timestamp and is selected to achieve a Least Recently Used (LRU) replacement policy.

The maximum number of elements in T and H will never be larger than C or M. In general C is significantly less than M, and the space complexity is reduced to $O(\max(C, M))$. Each operation on T will thus cost at most $O(\log(\max(C, M)))$, yielding a time complexity of $O(\frac{N}{P} \cdot \log(\max(C, M)))$.

VI. PARDA EVALUATION

Parda was implemented using MPI to enable execution on a distributed memory cluster. Instead of storing the data reference trace in a file, we use the Pin [10] tool to dynamically generate the memory address trace and use a Linux pipe to transfer the trace to feed process 0 of the parallel Parda execution. Figure 3 shows how the trace data generated by Pin is transfered to other processors. When process 0 receives the reference trace, it divides it into chunks and uses MPI to distribute chunks of the trace to the other processors. When processors receive their reference trace chunk from process 0, they apply the space optimized cache bound algorithm. After local processing, a merge is performed to complete processing of the slice. During this step, optimized local infinity processing is performed. Given that the highest rank processor has the latest trace elements before the merge, a communication reducing optimization of merging onto this processor is performed and it is renamed as the new virtual rank 0 process. After processing the entire trace, an MPI reduction operation is performed on the histogram to assemble the composite reuse distance histogram.

input : H, T, $hist$
  S_0^{C-1}, a sequence of data references
  l, the net count of local infinities
  B, the bounded storage size
output: H, T, $hist$, l
  loc_inf, a list of local infinities
let : loc_inf be a queue of local infinities

$loc\_inf \leftarrow \emptyset$
for $t \leftarrow 0$ **to** $C-1$ **do**
  $z \leftarrow S_0^{C-1}[t]$
  if $H(z) \neq \emptyset$ **then**
    $d \leftarrow$ distance$(T, H(z))$
    $T \leftarrow$ delete$(T, H(z))$
    $hist[d] \leftarrow hist[d] + 1$
  else
    if $l < B$ **then**
      $l \leftarrow l + 1$
      append(loc_inf, z)
    else if $|H| \geq B$ **then**
      $e\_min \leftarrow$ find_oldest(T)
      $H(e\_min.ref) \leftarrow \emptyset$
      $T \leftarrow$ delete$(T, e_min.ts)$
      $hist[\text{inf}] \leftarrow hist[\text{inf}] + 1$
    else
      $hist[\text{inf}] \leftarrow hist[\text{inf}] + 1$
    end
  end
  $T \leftarrow$ insert(T, z, t)
  $H(z) \leftarrow t$
end

Algorithm 7: Bounded reuse distance analysis algorithm.

A. Experimental Results

We conducted experiments to evaluate Parda's performance on a cluster with two 2.67 GHz quad-core Intel Xeon E5640 processors per node, running Redhat Linux Enterprise 5.4. The system is interconnected by Infiniband and uses the MVAPICH MPI implementation. We used benchmarks from the SPEC CPU2006 benchmark suite to evaluating Parda's performance.

Table IV shows the summary performance results for each benchmark, as well as the length of the trace (N) and the number of distinct references (M) in the trace. We present the execution time in seconds for two algorithms: i) the sequential tree-based algorithm [13] implemented using the efficient GLib hash table and a splay tree, and ii) Parda running on 64 processor cores (8 nodes), with a fixed pipe of 64Mw (256 MB), and a cache bound of 2Mw (8 MB). We also report the overheads associated with trace gathering: *Orig* represents the original execution time of the benchmark with no instrumentation; *Pin* is the time taken to run the benchmark modified with Pin instrumentation to generate the trace but without any reuse distance analysis; *Pipe* is the time taken for trace generation using Pin and feeding of all the data through the pipe to the processors but again without any analysis. This data is included to characterize inherent overheads for trace generation that are not attributable to the actual parallel Parda algorithm.

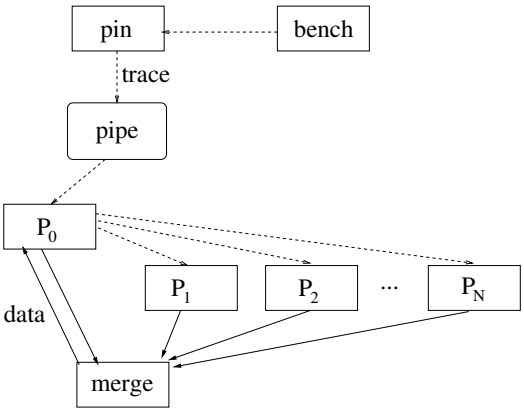


Figure 3. The Parda analysis framework.

From Table IV, we can see that Pin overhead (relative to unmodified benchmark execution time) ranges from a factor of 4.36 to 19.39, with an average 10.15. With inclusion of the overhead for pipe transmission of the data, the Pin overhead ranges from 7.88 to 30.47, with an average of 16.29 times the original benchmark runtime. The total analysis overhead for Parda ranges from 13.76 to 50.18, with an average of 28.48 times the original benchmark runtime. If we use average time as representative, for Parda with 64Mw pipe, 2Mw cache bound and 64 processors, reuse distance analysis overhead includes around 10x the benchmark runtime to generate the memory trace, around 6x to transfer the trace through a Linux pipe, and an additional 12x for the Parda algorithm processing (including total time of hash table and tree processing and all communication).

Figure 4 displays the variation of execution time for one of the benchmarks (MCF) as the number of processors is varied from 8 to 64, for cache-bound values ranging from 512Kw to 4Mw. Performance improves with decreasing cache size bound and the speedup from 8 to 64 processors is roughly 3.3x.

Figure 5 shows the performance impact of the cache bound parameter and the number of processors for all the benchmarks. In Figure 5(a), the cache bound is varied from 512Kw to 4Mw. We see that as the cache bound is increased, performance generally deteriorates slightly. This is due to the increased size of the tree and resulting cost of processing each reference. For some cases we see a slight reversal of this trend. This is due to the overhead of performing replacement in the tree and hash table data structures. When the cache bound is increased, the number of replacements

	Parameter		Overhead			Algorithm	
Benchmark	M	N	Orig	Pin	Pipe	Olken81	Parda
perlbench	23,857,981	11,194,845,654	5.93	106.43	180.71	7624.85	243.42
bzip2	11,425,324	8,311,245,775	5.41	59.13	86.88	6939.13	180.91
gcc	4,530,518	1,328,074,710	1.34	25.99	30.53	475.50	67.25
mcf	55,675,001	9,552,209,709	19.49	85.09	153.69	5898.61	268.29
milc	12,081,037	13,232,307,302	17.11	105.44	185.09	9746.86	365.60
namd	7,204,133	22,067,031,445	15.87	152.11	282.85	7936.16	431.55
gobmk	3,758,950	7,149,796,931	6.83	80.65	108.50	2798.21	186.21
dealII	31,386,407	66,801,413,934	39.59	522.24	674.06	20542.37	1250.43
soplex	18,858,173	3,432,521,697	3.87	32.25	52.24	187.19	102.59
povray	616,821	15,871,518,510	12.69	133.96	238.53	7503.35	307.91
calculix	10,366,947	2,511,568,698	2.18	24.45	42.18	1771.96	78.74
libquantum	570,074	1,700,539,806	2.43	13.56	26.93	715.78	58.81
lbm	53,628,988	48,739,982,166	43.47	339.75	674.09	26858.27	1211.35
astar	48,641,983	54,587,054,078	59.29	468.92	776.14	23275.32	1107.70
sphinx3	8,625,694	12,284,649,018	12.24	91.44	174.105	15331.22	290.51

Table IV
PARDA PERFORMANCE (SECONDS) M: TOTAL NUMBER OF DISTINCT ACCESSES, N: TOTAL NUMBER OF ACCESSES, ORIG: ORIGINAL PROGRAM RUNNING TIME, PIN: THE OVERHEAD TIME OF PIN TO GENERATE TRACE, PIPE: THE OVERHEAD TIME OF PIPE TRANSFER TRACE TO ANALYZER

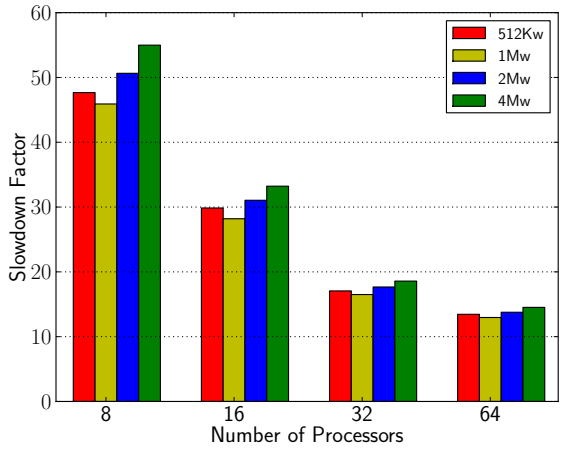


Figure 4. MCF performance data on a range of processors with fixed 64Mw pipe.

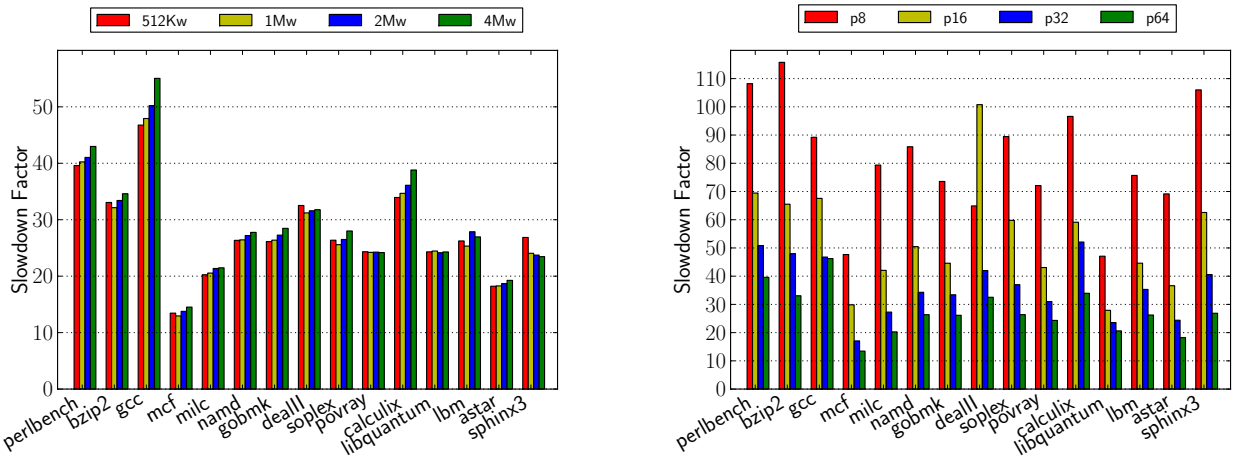
can decrease due to the increase in reuse distances in the data reference trace, leading to an improvement in performance. However, as seen in the MCF data in Figure 4, this trend can quickly reverse.

Figure 5(b) shows the variation in performance as the number of processors is varied from 8 to 64. We achieve an average speedup of over 3.5x across the benchmark suite. We can also see that performance gains are not as significant as more processors are added. This is in part due to the fact that the trace generation is sequential and represents a non-parallelizable overhead.

VII. RELATED WORK

In 1970, Mattson et al. were the first to define reuse distance (named LRU stack distance) [12]. Their initial algorithm used a list-based stack. As a stack is too slow for long reuse distances, many previous studies have explored different search data structures such as AVL trees to accelerate the analysis [2], [13], [18], [1]. Bennett and Kruskal used a vector and an m-ary tree to compute reuse distances and used hashing in a pre-pass [2]. Olken implemented the first tree-based scheme using an AVL tree [13]. Sugumar and Abraham showed that a splay tree has good memory performance for reuse distance analysis and their implementation has been widely available through the simulation toolset SimpleScalar [18]. Almasi et al. presented an algorithm that was based on a new search structure named interval tree [1]. In contrast to all of these previous works, which used sequential implementations, this paper focuses on accelerating reuse distance analysis by exploiting parallelism. Although our implementation is based on a splay tree and a hash table, our parallel algorithm can work with other data structures that implement stack as an abstract data type.

Since precise reuse distance calculation is very time consuming, many approximate approaches have been proposed to trade off accuracy for performance [4], [19], [15]. Ding and Zhong [4] introduced an approximate reuse distance analysis with an average computational complexity of $O(NloglogM)$ and an average storage cost of $O(logM)$. They also proved that their algorithms had bounded errors. Zhong and Chang [19], [22] demonstrated that sampling can be applied to reduce the time overhead of reuse distance analysis. Recently Schuff et al. [15] also applied sampling when analyzing traces on multicore processors. The algorithm developed in this paper significantly accelerates reuse distance analysis while still offering full accuracy. Moreover, our algorithm can be combined with approximate analysis techniques to further improve the performance of reuse distance analysis. It is worth noting that parallelism is also exploited in [15] but is analyzes multicore traces: each thread has its own trace that interacts with other traces through



(a) varying the cache size with fixed 64 processes and 64Mw pipe

(b) varying the number of processors with fixed cache size 512Kw and 64Mw pipe

Figure 5. Analysis of Parda overhead across a range of benchmarks in the SPEC CPU2006 suite.

synchronization points. In contrast, our approach is able to analyze a single trace in parallel.

As the majority of the cache designs are based on the LRU (Least Recently Used) replacement policy or its approximations, reuse distance is an effective metric that summarizes the data locality of a given trace, without introducing lower-level cache details such as latencies, associativities and the actual replacement algorithm. A number of studies have used reuse distance in performance analysis and optimization at different granularities. At the whole-program level, Zhong et al. proposed an algorithm that generated a parameterized model of whole-program cache behavior based on reuse distance, and used this model to predict the miss rate for arbitrary data input set sizes [20]; also with a parameterized model based on reuse distance analysis, Marin and Mellor-Crummey predicted the L1, L2 and TLB cache miss counts across architectures for scientific programs [11]; Shen et al. used reuse distance to characterize and predict program phases [16]. At the data object level, Zhong et al. used reuse distance to model reference affinity and demonstrated its application in data layout optimization [21]; Lu et al. analyzed locality patterns for individual data objects using reuse distance and partitioned the last-level cache between objects to improve program performance [9]. At the instruction level, Fang et al. showed how reuse distance can be applied in identification of performance-critical instructions and memory disambiguation [5], [6]. Reuse distance has also been used for optimization at the operating system level. Cascaval et al. used reuse distance information to manage superpages [3] and Jiang and Zhang applied reuse distance (referred to as recency in the paper) to improve disk buffer performance. [7]. Although our demonstration of the parallel reuse distance algorithm in this paper has been through whole-program reuse distance analysis, our parallel algorithm can be applied for different reuse distance-based analyses and optimization techniques. We expect that with the advent of manycore processors and accelerators to commodity systems, parallel reuse distance analysis will benefit a larger class of systems and applications, in particular those that rely on online analysis, such as cache sharing and partitioning.

VIII. Conclusions

In this paper we presented Parda, a parallel algorithm that can greatly reduce the time required to perform reuse distance analysis. The primary contribution of this paper lies in the identification of parallelism present in data reference trace analysis and in harnessing this parallelism to accelerate reuse distance analysis. To our knowledge, our work represents the first effort to parallelize this valuable performance analysis tool. Results indicate that Parda online analysis with 64 cores is able to achieve performance within a factor of 13 to 50 times the original performance for a range of applications.

Acknowledgements

This work was supported in part by the National Science Foundation through award 0904549 and the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] George Almási, Călin Caşcaval, and David A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*, MSP '02, pages 37–43. ACM, 2002.

[2] B.T. Bennett and V.J. Kruskal. LRU stack processing. *IBM Journal for Research and Development*, pages 353–357, July 1975.

[3] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 339–349. IEEE Computer Society, 2005.

[4] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. *Programming Language Design and Implementation (PLDI)*, pages 245–257, 2003.

[5] C. Fang, S. Carr, S. nder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. *Proceedings of the 2004 workshop on Memory System Performance*, pages 60–68, 2004.

[6] Changpeng Fang, Steve Carr, Soner Onder, and Zhenlin Wang. Instruction based memory distance analysis and its application. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 27–37. IEEE Computer Society, 2005.

[7] Song Jiang and Xiaodong Zhang. Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance. *IEEE Trans. Comput.*, 54:939–952, August 2005.

[8] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In Rajiv Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 264–282. Springer, 2010.

[9] Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 246–257. IEEE Computer Society, 2009.

[10] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[11] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 2–13. ACM, 2004.

[12] R.L. Mattson, J. GECSEI, D. SLUTZ, and I. L Traiger. valuation techniques for storage hierarchies. *IBM Syst*, pages 78–117, 1970.

[13] F. Olken. Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies. *Technical Report LBL-12370, Lawrence Berkeley Laboratory*, 1981.

[14] Pavlos Petoumenos, Georgios Keramidas, Håkan Zeffer, Stefanos Kaxiras, and Erik Hagersten. Modeling cache sharing on chip multiprocessor architectures. In *IISWC*, pages 160–171. IEEE, 2006.

[15] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.

[16] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 165–176. ACM, 2004.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32:652–686, July 1985.

[18] R. Sugumar and S. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[19] Y. Zhong, X. Shen, and C. Ding. Program Locality Analysis Using Reuse Distance. *ACM Transactions on Programming Languages and System*, 31(6), 2009.

[20] Yutao Zhong, Steven G. Dropsho, and Chen Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 79–. IEEE Computer Society, 2003.

[21] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 255–266. ACM, 2004.

[22] Yutao Zhong and Chang Wentao. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*, ISMM2008. ACM, 2008.