Featherlight Reuse-distance Measurement

Qingsen Wang
College of William & Mary
Email: qwang@cs.wm.edu

Xu Liu
College of William & Mary
Email: xl10@cs.wm.edu

Milind Chabbi
Scalable Machines Research
Email: milind@ScalableMachines.org

***Abstract*—Data locality has a profound impact on program performance. *Reuse distance*—the number of distinct memory locations accessed between two consecutive accesses to the same location—is the de facto, *machine-independent* metric of data locality in a program. Reuse distance measurement, typically, requires exhaustive instrumentation (code or binary) to log every memory access, which results in orders of magnitude runtime slowdown and memory bloat. Such high overheads impede reuse distance tools from adoption in long-running, production applications despite their usefulness.**

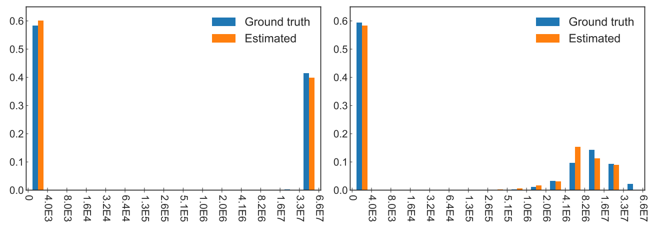
We develop RDX, a lightweight profiling tool for characterizing reuse distance in an execution; RDX typically incurs negligible time (5%) and memory (7%) overheads. RDX performs no instrumentation whatsoever but uniquely combines hardware performance counter sampling with hardware debug registers, both available in commodity CPU processors, to produce reuse-distance histograms. RDX typically has more than 90% accuracy compared to the ground truth. With the help of RDX, we are the first to characterize memory performance of long-running SPEC CPU2017 benchmarks.

***Keywords*-Reuse distance; locality; hardware performance counters; debug registers; profiling.**

I. Introduction

Maintaining data locality in programs remains as pertinent as ever because of the disparity between the cost of fetching data from different levels in the memory hierarchy and the cost of performing computation over the data. The evolution of the memory hierarchy—deep off-chip L4 caches [1], large on-chip L3 caches [2], multi-socket and multi-node NUMA architectures [3], and heterogeneous memories [4]—only exacerbates the need to maintain good locality in programs.

`Reuse distance` is the de facto software metric to quantify data locality in an execution. Reuse distance, also known as *stack distance* [5], is defined as the number of *distinct* memory elements accessed between the current memory access (*reuse*) and the previous memory access to the same memory element (*use*). For example, given a sequence of memory accesses: a_1, b_1, c_1, b_2, a_2, where the subscripts represent the access number of the same location, the reuse distance for memory location a at the access instance a_2 is 2 since two other elements b and c were accessed between consecutive access to a . If the reuse distance of a memory location is larger than the cache size, a capacity cache miss is guaranteed even in the absence of conflict misses in a typical LRU (least recently used) cache. By varying the analysis granularity of a single memory word or a cache line, reuse distance can quantify both temporal and spatial data locality.



(a) lbm (reference input) **(b) milc (reference input)**

Fig. 1: The real and estimated stack reuse distance histograms of SPEC CPU2006 `lbm` and `milc`. The x-axis is in logarithm-scale ranges of reuse distances; the y-axis is the fraction of reuse pairs falling in specific distance ranges over the total number of reuse pairs.

Reuse distance is often presented as a histogram with bins representing different reuse distance ranges. Fig. 1 shows the reuse distance histograms for two SPEC CPU2006 benchmarks—`lbm` and `milc`, where the x-axis represents a series of ranges and the height of each bar indicates the percentage of reuse distances falling into the corresponding range. Fig. 1 also shows that the reuse distance histograms can be accurately estimated by the lightweight tool, RDX, proposed in this paper.

Collecting the reuse distance for the entire program execution—*whole program reuse distance*—provides deep insights into a program's locality characteristics. Whole program reuse distance enables various studies: performance prediction [6–8], cache simulation [9, 10], program phase prediction [11], processor caching [12–18], per-instruction miss rate prediction [19], profiling and code tuning [20–26], power characterization [27], to name a few.

There exist a number of tools [5, 28–32] that provide reuse distances profiles for entire program executions. These tools instrument every load and store operation via a compiler or binary rewriter to obtain the effective memory address at runtime. At runtime, an analysis routine logs the address into an augmented search tree [5]. Upon each memory access, these tools check the previous access to the same address and count the number of unique memory addresses touched in between to record an instance of reuse distance. Each instance, based on its precise reuse distance, is accumulated into a bin representing a range of reuse distances (typically log scale). Finally, bins are presented as a histogram where each bar the bin represents the fraction of total reuse-distances falling into its range. Although these tools provide detailed analyses, their exhaustive instrumentation and logging mechanisms incur hundreds of times slowdown and consume enormously extra memory [21, 33] preventing their use on long-running, production programs.

Prior work has explored various means to reduce the overhead of computing reuse distances. Some approaches [20, 22, 23, 34, 35] adopt sampling to monitor a subset of memory accesses, while others [36, 37] exploit multi-cores for acceleration. Other approaches [38, 39] collect less accurate reuse distances to reduce the measurement overhead. None of these schemes eliminate the heavyweight instrumentation; hence, the overhead remains nontrivial—more than $5\times$. Production codes with service-level guarantees are intolerant to overheads higher than just a few percents and timeout if not completed in a specified time; thus cannot be monitored with these heavyweight techniques. Modern micro-service architectures have complex inter-service dependencies making it cumbersome, if not impossible, to run in a simulator or instrumented environment.

To address the high overhead seen in instrumentation-based reuse-distance measurement tools, we develop RDX—a featherlight profiler that collects reuse information without software instrumentation of memory accesses. RDX leverages event-based sampling [40–42] enabled by the hardware performance monitoring units (PMU) to sample memory accesses (point of use) and utilizes the hardware debug registers [43, 44] to trap on the point of reuse. RDX counts the number of memory accesses elapsed between the use and reuse, which is known as *time distance*. RDX transforms the time distance profiles into stack distance histograms via a well-known algorithm [38].

In summary, we make the following contributions:

- Develop a reuse-distance profiler—RDX—that integrates PMU event-based sampling and debug registers to produce reuse-distance profiles. RDX reduces the reuse-distance profiling overhead to a few percents, so that it can be used on every code check-in in development environments to identify data-locality regressions.
- Demonstrate that PMU-supported *sampled time-distance* profiles provide high-accuracy stack reuse distance profiles when compared to heavyweight instrumentation tools.
- Offer a solution to reuse distance histograms that works on fully optimized binaries, which needs no instrumentation and no special hardware extension to commodity CPUs.
- Characterize the data locality of the newly introduced SPEC CPU2017 benchmark suite [45], which has many long-running programs. The insights complement earlier studies [46–48].

The rest of the paper is organized as follows. Section II offers an overview of the methodology employed by RDX. Section III provides the background needed to understand the details of RDX. Section IV provides the design and implementation of RDX. Section V evaluates RDX on a real CPU architecture. Section VI explores different use cases including the data locality of SPEC CPU2017 and guided locality optimization with RDX. Section VII reviews the prior work and distinguishes our approach. Section VIII presents our conclusions.

II. RDX Methodology

Shen et al. [38] developed the notion of *time reuse*, which is the number of *total* memory accesses performed between two consecutive accesses to the same location. The time reuse relaxes the *distinct* nature of stack-reuse[1]. In the access sequence a_1, b_1, c_1, b_2, a_2, the time reuse distance of a at a_2 is four, since there are four memory accesses (a_1, b_1, c_1, b_2) between two consecutive accesses to a.

While collecting the stack-reuse profiles has a very high overhead, collecting the time reuse profiles has a lower overhead. If M is the number of distinct elements accessed in a program, while the stack reuse incurs an $O(logM)$ cost (via a balanced binary search tree) on every memory access [39], the time reuse incurs only $O(1)$ cost. The constant overhead for time reuse comes by maintaining a shadow-memory for each memory address, which stores the system-wide memory access count at the time of accessing it. The difference between the current access count and the previous access count is the time reuse for the current instance of memory access, and it can be inserted into a time-reuse histogram in constant time.

Furthermore, Shen et al. [38] also devised an elegant, offline algorithm to convert a time-reuse histogram into a stack-reuse histogram with a very high (99%) accuracy. We only offer an intuitive idea of this transformation here. Consider a memory access sequence $\mathtt{a}_1$ `X X X X` $\mathtt{a}_2$, where `X` is any address other than `a`. The time distance of `a` at $\mathtt{a}_2$ is 5, but we have no idea how many other distinct elements were accessed in between. If three time reuses are of distance 1 and one time reuse is of distance 5 in the entire sequence, we definitely know there should be only one distinct element between consecutive accesses to `a`, and hence $\mathtt{a}_2$'s reuse distance is 1. While an exhaustive enumeration of the possibilities is untenable, Shen et al. models it as a Bernoulli process assuming that the accesses are *independent*. We delegate the details of this algorithm to Appendix A.

Shen et al.'s scheme, however, does not avoid instrumenting every load and store and hence incurs 20-40$\times$ runtime overhead. The key insight we exploit in RDX is

[1]Stack reuse is 0-based (e.g., a_2's stack reuse distance in the sequence a_1, a_2 is zero); time reuse is 1-based (e.g., a_2's time reuse distance in the sequence a_1, a_2 is one).

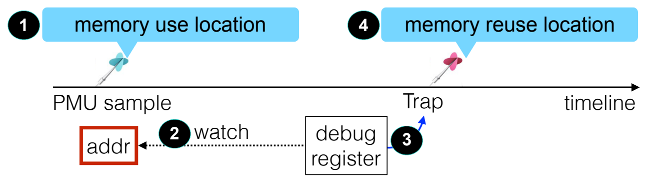


Fig. 2: RDX's sampling scheme for reuse pairs. ① PMU samples a memory access that touches memory address `addr`, referred to as use. ② In the PMU sample handler, RDX arms a debug register to monitor the program's next access to `addr`. ③ The debug register traps on the next load or store to `addr`, referred to as reuse. ④ RDX counts the number of memory accesses between the use and reuse, which is the time distance for the sampled use-reuse pair.

that one need not monitor every load and store to derive the time-reuse distance profiles—sampling memory addresses accessed by the program suffices. This is a key departure of our approach from Shen et al.'s scheme. Furthermore, for sampling, we perform no instrumentation whatsoever but instead employ the advancements in commodity CPUs to sample performance counters, which expose the effective memory address accessed by the program in each sample.

The problem now reduces to identifying the count of memory accesses performed between two consecutive accesses to a sampled address. Existing PMUs offer no facility to count the number of memory accesses between two consecutive accesses to the same address. It is imprudent to expect such details from any PMU. We employ yet another hardware feature to circumvent this limitation. Hardware debug registers, available in commodity CPUs, offer the facility to trap execution on accessing a given memory location. Thus, the address of a PMU sample, when used to trap using hardware debug registers offers a window into consecutive accesses to the same memory location. The number of memory accesses elapsed between these two points is the *time reuse* distance.

Empirically, we show that time-reuse profiles obtained with this technique are ∼97% accurate compared to the ground truth time-reuse histograms obtained via exhaustive instrumentation. With such high-fidelity time-reuse histograms, we reconstruct the stack-reuse histogram by applying the Shen et al.'s statistical model and produce high fidelity (∼93% accurate) stack-reuse histograms. Fig. 2 depicts the overall scheme used in RDX. In addition to collecting the reuse profiles (histograms), RDX attributes every use-reuse pair to the full calling context and in turn to the application source code for a post-mortem inspection. RDX has a built-in GUI to sort and visualize hot reuse pairs in their full calling contexts to aid developer inspection.

III. Background and Terminology

In this section, we introduce the background knowledge on PMUs and debug registers available in commodity CPUs, which form the backbone of our work. We also describe a few essential terms used in the rest of this paper.

Hardware Performance Monitoring Unit (PMU): PMU in a CPU offers a programmable way to count hardware events such as loads, stores, CPU cycles, to name a few. PMUs can be configured to trigger an overflow interrupt on reaching a threshold number of events. A profiler, running in the address space of the monitored program in the case of RDX, handles the interrupt and attributes the measurement "appropriately" to the execution context. We refer to a PMU interrupt as a "sample." PMUs are per CPU core and virtualized by the operating system (OS) for each OS thread.

Intel's Precise Event-Based Sampling (PEBS) [40] facility offers the ability to inspect the effective address (EA) accessed by the instruction on an event overflow for certain kinds of events such as loads and stores. This ability to extract the effective address is often referred to as "address sampling", which is a critical building block of RDX. AMD Instruction-Based Sampling (IBS) [41] and PowerPC Marked Events (MRK) [42] offer similar capabilities.

Hardware debug registers: Hardware debug registers [43, 44] enable trapping the CPU execution for debugging when the PC reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can program debug registers with different addresses, widths, and conditions (`W_TRAP` and `RW_TRAP`) that will cause the CPU to trap on reaching the programmed conditions. Today's x86 processors have four debug registers; POWERPC processors provide one programmable debug register.

Linux perf_events: Linux offers a standard interface to program and sample PMUs and debug registers using the `perf_event_open` system call [49] and the associated `ioctl` calls. The Linux kernel can deliver a signal to the specific thread whose PMU event overflows or debug register traps. The user code can (1) `mmap` a circular buffer into which the kernel keeps appending the PMU data on each sample and (2) extract the stack trace (calling context) on each signal.

Terminology: A *watchpoint* is a software abstraction of a debug register to monitor data access. An address is *monitored* if we have set a watchpoint at that address. A watchpoint can be set to trap on write only (`W_TRAP`) or trap on read-or-write (`RW_TRAP`). A *watchpoint exception* (aka trigger) is a synchronous CPU trap caused when an instruction accesses a monitored address. A *PMU sample* is a CPU interrupt caused when an event counter overflows. Both PMU samples and watchpoint exceptions are handled via Linux signals. In the rest of this paper, any unqualified use of the term "reuse distance" means "stack reuse distance", we always refer to "time reuse distance" with qualification.

IV. Design and Implementation

RDX samples memory accesses via PMU counters configured to count every load and store operation. PMU generates an interrupt each time a preconfigured threshold number of memory accesses elapse.

The use point: On a PMU counter overflow (interrupt), RDX obtains the effective address, say $\mathcal{M}$, involved in the overflow—this is the *use* point. RDX records the monotonically growing PMU counter value (say V_{use}) at this point. RDX also records the current calling context C_{use} for later use. To detect the *reuse* point, RDX sets a watchpoint at $\mathcal{M}$ and lets the execution resume.

The reuse point: When the program accesses the same location $\mathcal{M}$ again (the *reuse* point), the watchpoint traps. RDX records the PMU counter value (say V_{reuse}) once again at this point. $R = V_{reuse} - V_{use}$ is the number of memory accesses elapsed between two consecutive accesses to $\mathcal{M}$, which is the time reuse distance of this instance. We increment the count of the time-reuse histogram bin that R belongs to. Additionally, RDX collects the calling context C_{reuse} and attributes an instance of time reuse to the calling context pair $\langle C_{use}, C_{reuse}\rangle$.

The profiling continues throughout program execution, which collects many reuse instances along with their time distances. At the program termination, RDX converts the sampled time-distance profiles into stack-distance profiles following Shen et al.'s model [38]. The random sampling of addresses acts like a monte-carlo experiment; with sufficient samples, the fraction of reuse distance in different bins of a histogram closely matches the ground truth.

Since RDX uses PMU for address sampling and debug registers for address monitoring, there is no need to instrument code or perform any analysis on every memory access. Overhead is incurred only in the PMU sample interrupt handler and debug register trap handler, which is controlled by the PMU sampling frequency. Although we explained the scheme as if a single counter could count both loads and stores, typically, different counters count them in commodity CPUs. In such situations, we need to read both counters at each use and reuse point to count the number of elapsed memory accesses correctly.

Since reuse can happen only on memory access instructions and every memory access instruction is sampled at a frequency proportional to its occurrence, transitively, we detect reuse at a frequency proportional to their occurrence if we had *infinite* debug registers.

A. Working with limited number of debug registers

Hardware can monitor only a small number of addresses at a time since they have only a few debug registers. The scenario where many PMU samples interrupt between the use and reuse accesses to the same memory location complicates matters.

Consider the reuse example in Listing 1. Assume the loop index variables `i`, `j`, and the scalars `t`, `m` are in registers, the sampling period is 10K memory accesses, and the number of debug registers is one. The first sample happens in the `i` loop when accessing `array[10K]`. RDX sets a watchpoint to monitor `&array[10K]` since a debug register is available.

```
for(int i = 1; i <= 100K; i++){
  t += array[i];
}
for(int j = 1; j <= 100K; j++){
  m += array[j];
}
```

Listing 1: Long distance reuses: All four watchpoints are armed when sampling at 10K memory accesses in the first four samples taken in the `i` loop. A naive replacement does not trigger a single watchpoint due to many samples taken in the `i` loop before reaching the `j` loop. RDX ensures each sample equal probability to survive.

The second sample happens when accessing `array[20K]`. Since the watchpoint armed for address `&array[10K]` is still active, there is no room to monitor `&array[20K]`.

Naively, one may replace the previously set watchpoint (`&array[10K]`) with `&array[20K]`. This scheme, however, does not detect any reuse in this code. When the `j` loop starts executing, the only active watchpoint would be the last sampled address `&array[100K]` in the `i` loop. The PMU keeps delivering samples in the `j` loop as well. At `j=10K`, this scheme would replace the last watchpoint `&array[100K]` with `&array[10K]`, which is not accessed again. At the end of the `j` loop, not a single watchpoint would have triggered and hence no reuse could be detected.

Monitoring a new sample may help detect a new, previously unseen reuse whereas continuing to monitor an old, already-armed address may help detect a reuse separated by many intervening operations. We should detect both. However, we cannot predict when in the future a watchpoint may trap, if at all. A slightly smarter strategy is to flip a coin to decide whether or not to set a watchpoint for the newest sample. This strategy also fails because the survival probability of an older sample is minuscule if the distance between consecutive accesses to the same memory location is significantly larger than the sampling period.

We employ reservoir sampling [50, 51], which strikes a balance between new versus old by choosing among the previously accessed addresses without any bias. Consider the case of a single debug register to easily understand the reservoir sampling scheme. The first sampled address, M_1, occupies the debug register with 1.0 probability. The second sampled address, M_2, overwrites the previously armed watchpoint with $1/2$ probability and retains the old one with $1/2$ probability. The third sampled address, M_3, overwrites the previously armed watchpoint with $1/3$ probability and retains the old one (either M_1 or M_2) with $2/3$ probability. The k^{th} sampled address M_k since the last time a debug register was empty, replaces the previously armed watchpoint with $1/k$ probability. At the end of the k^{th} sample, the probability P of monitoring any sampled address M_i, $1 \le i \le k$, is the same.

Any time a watchpoint traps, RDX disarms the watchpoint and resets its reservoir probability to 1.0. Obviously, if every watchpoint triggers before the next sample, we will monitor

```
 1 for(int i = 1; i <= 100K; i++){
 2   ... = array[i]; // sparse reservoir samples
 3 }
 4 for(int j = 1; j <= 100K; j++){
 5   ... = array[j]; // sparse reservoir samples
 6 }
 7 for(int k = 1; k <= 100K; k++){
 8   ... = array[k]; // dense reservoir samples
 9   ... = array[k]; // dense reservoir samples
10 }
```

Listing 2: Long-distance reuse instances between loop `i` and `j` may be under counted (assuming a sampling period of 10K), which causes disproportionate attribution compared to short-distance reuses in loop `k`.

every address seen in every sample. The sampling scheme maintains only a *count* of previous samples (not an access *log*), which consumes a constant amount of memory.

When there are more than one debug registers, RDX maintains an independent replacement probability for each debug register. When a sample occurs, if a debug register is unused, RDX arms the free debug register and decrements the replacement probability of all other armed debug registers. However, if all debug registers are occupied, RDX visits every debug register i and attempts to replace it with P_i probability. The process may succeed or fail in replacing a debug register with a new sample, but it gives a new sample N chances to remain in a system with N watchpoints. Whether success or failure the P_i of each in-use debug register is updated after a sample. The order of visiting the debug registers is randomized for each sample to ensure fairness. The evaluation section shows the robustness of our scheme by varying the number of debug registers.

Proportional attribution to avoid attribution bias: Consider the program shown in Listing 2. Assuming a sampling period of 10K, the reservoir sampling retains only a few samples between loop `i` till loop `j`, since the time reuse distance (100K) is larger than the sampling period. Hence, we will record only as many reuses in loop `j` as the capacity of the reservoir (e.g., only one). However, in the `z` loop, a sampled address traps almost immediately since the time reuse distance (unit distance) is shorter than the sampling period. Thus, there will be as many reuses detected as the number of samples (e.g., ten in this case). In reality, there is *equal quantity* of short and long time reuses in the execution. Thus, reservoir sampling leads to a disproportionate attribution based on whether only a subset of sampled addresses are monitored (when the reservoir is full at the sample point) or all sampled addresses are monitored (reservoir is not full at the sample point).

We use the context-sensitive scaling scheme devised by Wen et al. [51] that corrects this accounting problem. In a nutshell, it uses the heuristic that the code behavior is typically the same in a calling context, hence when a watchpoint traps, rather than recording only one instance of the reuse, we scale it by the number of samples (≥ 1) taken at the calling context of the use point starting from the time the watchpoint was armed. Thus in the case of Listing 2, when a debug register traps on line 5, we refer back to the number of samples taken at line 2, which is ten, and attribute ten, reuses between these line pairs. This is one of the reasons for collecting the calling context C_{use} at the point of a sample. This scheme proves to be highly effective (see Section V).

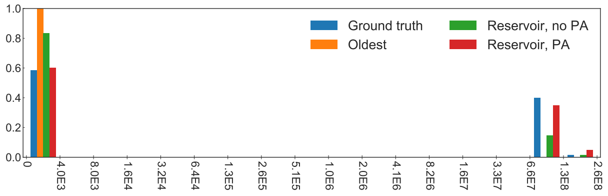


Fig. 3: A comparison of ground truth vs. different sampling strategies over SPEC CPU2006 `lbm`. Replacing the "oldest" watchpoint is inaccurate. Replacement using the reservoir sampling but *without* proportional attribution (PA) enhances accuracy but still falls short. Reservoir sampling *with* proportional attribution achieves near-perfect accuracy.

Fig. 3 compares the reuse distance histograms of `lbm` from SPEC CPU2006 obtained with different sampling strategies.

B. Concrete Implementation using HPCToolkit

We implement RDX atop the open-source HPCToolkit [52] performance analysis tools suite. HPCToolkit works on multi-lingual, multi-threaded, and multi-process, fully optimized application binaries on multiple programming models such as MPI, OpenMP, and pthreads. On a PMU sample, HPCToolkit's profiler walks the sampled thread's call stack via an online binary analysis [53]. It, then, attributes the measurements to the sampled call path. HPCToolkit introduces negligible runtime overhead ($\sim$3%) and consumes only a few megabytes of memory space for its metric data when sampling at $\sim$200 samples/second/thread [53]. HPCToolkit further aggregates all the profiles from different threads/processes and associates them with program source code in a GUI for intuitive analysis. We elaborate on different components of RDX below.

RDX's online profiler: RDX's profiler loads the monitoring library into the target application's address space at link time for statically linked executables or at runtime using `LD_PRELOAD` [54] for dynamically linked executables. As the target application executes, RDX's profiler manages PMUs and hardware registers to record reuse pairs and captures full calling contexts for both use and reuse instances. To minimize synchronization overhead during execution, each thread records its own profile.

Although RDX can sample any precise PMU event to set a watchpoint, on Intel processors, typically, we use `MEM_UOPS_RETIRED:ALL_STORES` to sample memory stores and `MEM_UOPS_RETIRED:ALL_LOADS` to sample memory loads. All of these events offer the effective memory address accessed in a sample along with the program counter.

Monitoring stack addresses in the target application is

Table I: The evaluation platform.

CPU	18-core Xeon E5-2699 v3 (Haswell) @ 2.30GHz
Cache	32KB L1D, 32KB L1I, 256KB L2, 45MB L3
Misc	128GB DDR, `linux 4.8`, `gcc-5.4.1 -O3`

tricky because the frames of RDX's sample/trap handler can overwrite the stack location and cause undesired debug register trap. We avoid this problem by establishing a separate signal-handler stack frame for both PMU signal handler and watchpoint exception handler using the Linux `sigaltstack` facility [55]. The `sigaltstack` facility allows each thread in a process to define an alternate signal stack in a user-designated memory region. We use an alternate stack to handle PMU and watchpoint signals. All other signals continue to use the default stack unless specified otherwise by the application.

Post-mortem analysis: RDX merges profiles from different threads/processes and plots the reuse histogram. RDX converts the time reuse distance histograms to the stack reuse distance histograms according to Shen et al.'s model, which typically requires a few seconds to process the data.

V. Evaluation

In this section, we evaluate RDX for its accuracy and overhead on the `ref` inputs of SPEC CPU2006 [56] suite. Our evaluation machine configuration is shown in Table I. As an exception, we compile `perlbench` with `gcc-4.8.5 -O3` because it does not compile with `gcc-5.4.1`. We sample memory loads and stores each at sampling periods 500K, 1M, 5M, and 10M to evaluate RDX's accuracy and overhead. A lower value of *sampling period* implies taking samples *more frequently*, which obviously incurs more overhead. To be specific, a sampling period of 500K means two PMU counters (one each for loads and stores) are concurrently running, and each one traps once it reaches its 500K threshold; this means if we have a balanced loads and stores, we will be taking an interrupt every 250K memory accesses. We collect measurements five times for each benchmark under each sampling period and report the average results along with the error bars. The measurement error for accuracy and overhead is negligible ($< 0.1\%$) and are usually invisible in the figures. Since some SPEC CPU benchmarks have multiple input files, we distinguish the same benchmark with different inputs with numerical suffixes such as `gcc-1`, `gcc-2`, etc. The actual mapping of these names to inputs is presented in Appendix B.

A. RDX Accuracy

Baseline. Evaluating RDX accuracy requires comparing with the ground truth. For collecting the baseline (ground truth), we initially considered the publicly available stack reuse distance profiler *Loca* [29], which instruments every load and store operation using Pin [57]. However, we encountered limitations with *Loca*—it does not correctly handle programs with more than 4GB memory footprint, and it does not produce 100% accurate results because it applies certain approximations [39]. Hence, we developed our own Pin-based tool that eliminated the limitations of *Loca*.

Our ground-truth instrumentation tool has two components—a time reuse distance profiler and a stack reuse distance profiler. Both employ a shadow-memory [58] where the timestamp (previous access count) of each byte is recorded. The time reuse distance is simply a constant-time operation per memory access. The stack reuse, in addition to the shadow memory, employs a balanced binary tree, which is accessed (looked up, deleted from, and inserted into) on each memory access. The tool produces three kinds of histograms: 1) a time-reuse histogram, 2) a stack-reuse histogram, and 3) a stack-reuse histogram derived from the time-reuse histogram, similar to [38]. We omit the implementation details of this tool for brevity.

Reuse histograms. We present all reuse histograms on a log base 2 scale and the bins fall in the following ranges: $[0, 4K), [4K, 8K), [8K, 16K) \cdots, [1G, 2G)$. All histograms have 20 bins. Reuse distances of larger than 2GB get assigned to the last bin. We rarely encountered (time or stack) reuse distances of such large magnitude; one exception is `603.bwaves_s`, whose longest time distance spike is in the bin of $[2G, 4G)$. At the profiling time, there is no limitation on the number of bins. The 20-bin limit is only for consistent visualization and comparison. The height of the histogram bins is normalized by the total number of reuses observed by RDX; hence, each bin represents the fraction of total reuses belonging to that bin. The sum of all bins adds up to 1.0.

Accuracy metric. Given two histograms H and $\hat{H}$, we compare their similarity with the following formula.

$$\mathcal{S} = 1 - \frac{\sum_{i=1}^{n} |B_i - \hat{B}_i|}{2} \quad (1)$$

B_i and $\hat{B}_i$ are the fraction of reuses that fall in the i^{the} bin of H and $\hat{H}$, respectively, and n is the total number of bins. When H is the ground truth and $\hat{H}$ is the estimated value, $\mathcal{S}$ measures accuracy of $\hat{H}$ with respect to H. $\mathcal{S}$ is in the range $[0, 1]$, and $\mathcal{S} = 1.0$ means perfect matching between the two histograms. If there are only two bins and if the ground truth attributes all its reuses to its bin-1 and the estimation attributes all its reuses to its bin-2, $\mathcal{S} = 1 - |\frac{1.0-0.0}{2}| + |\frac{0.0-1.0}{2}| = 0$; hence zero similarity.

The $\mathcal{S}$ metric is the same as the one used by Shen at al. [59]. $\mathcal{S}$ is a strict metric; it does not tolerate off-by-one binning in case of measurement errors. For example, if the estimated reuse distance is slightly larger (smaller) than a bin's range, and hence gets attributed to the next (previous) bin, $\mathcal{S}$ metric may severely penalize their similarity. When necessary, we present another metric $\widehat{\mathcal{S}}$, which computes a sliding window average over two adjacent bins to mitigate

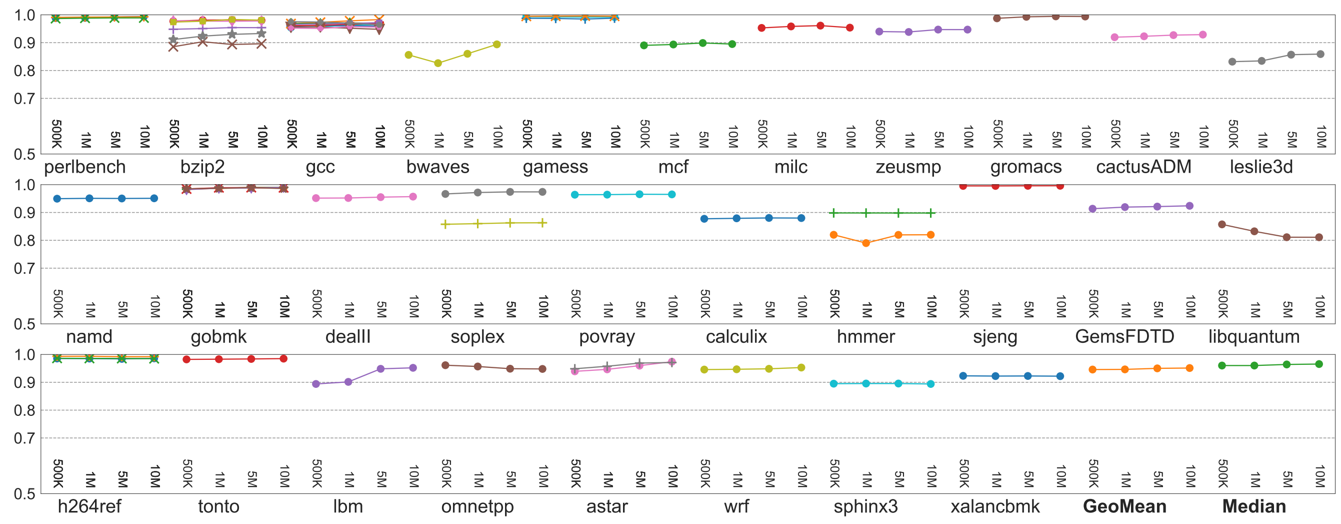


Fig. 4: RDX time reuse distance accuracy for SPEC CPU2006 benchmarks at 500K, 1M, 5M and 10M sampling periods. One benchmark may have multiple lines due to runs with different inputs.

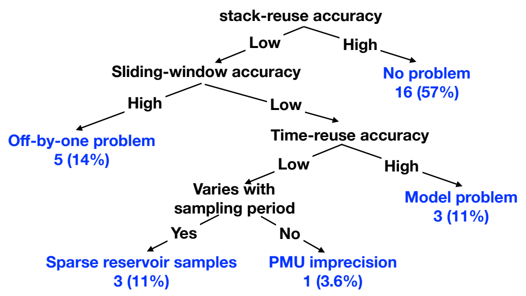


Fig. 5: A decision-tree procedure to investigate low accuracy in stack reuse distance estimates. We treat lower than 90% as low accuracy. The leaf nodes show the number (%) of SPEC CPU benchmarks that fall in that category.

the off-by-one error.

$$\hat{\mathcal{S}} = 1 - \frac{\sum_{i=1}^{n-1} |Avg(B_i, B_{i+1}) - Avg(\hat{B}_i, \hat{B}_{i+1}|)}{2} \quad (2)$$

In the rest of this section, we first evaluate the accuracy of time distance histograms generated by RDX and then the stack distance histogram. We employ a systematic approach to investigate the accuracy of the benchmarks using a strategy shown in Fig. 5.

1) RDX accuracy for time distance histograms: Fig. 4 shows the accuracy metric $\mathcal{S}$ of time distance histogram produced by RDX for SPEC CPU2006 benchmarks under different sampling periods. Overall, RDX yields high accuracy with a median of higher than 96%; the accuracy is insensitive to the sampling period for most benchmarks. The high accuracy of time distance histograms is the foundation to construct the accurate stack distance histograms. There are some outliers with accuracy less than 90%, which we discuss in the next subsection.

Fig. 6 evaluates the accuracy of time reuse histograms at 5M sampling period by (1) varying the number of debug registers from one to four, and (2) disabling the proportional attribution. The setting that disables the proportional attribution uses 4 debug registers. From the figure, we can infer the following: (1) the number of debug registers has minuscule impact on the accuracy, which validates the strength of the reservoir sampling, and (2) proportional attribution gives significant accuracy boost for some benchmarks, e.g., `bwaves`, `mcf`, `milc`, `zeusmp`, `soplex-2`, `GemsFDTD`, and `lbm`.

2) RDX accuracy for stack distance histograms: Fig. 7 shows stack distance accuracy of RDX compared to the ground truth on SPEC CPU2006 benchmarks at different sampling periods. RDX yields a median accuracy of 90%. Some benchmarks suffer from lower accuracy for the following reasons.

- *Off-by-one problem.* An off-by-one problem happens when a reuse bin gets attributed to an adjacent bin in the histogram. The ground truth numbers that are on the boundary of two bins, when estimated during sampling and reconstructed through the modeling, can sometimes get attributed to the adjacent bin. Since sampling has measurement errors and skid effects, this is the most common cause for the cases where the accuracy is below 90%. Benchmarks suffering from this problem have superior (higher) sliding-window metric $\hat{\mathcal{S}}$, compared to the stricter $\mathcal{S}$ metric. Fig. 8 shows how the long-distance reuses are bin shifted by one left for SPEC CPU2006 `soplex` ($\hat{\mathcal{S}} = 0.74$) benchmark. `bwaves` ($\hat{\mathcal{S}} = 0.73$), `leslie3d` ($\hat{\mathcal{S}} = 0.89$), `sphinx3` ($\hat{\mathcal{S}} = 0.78$), `GemsFDTD` ($\hat{\mathcal{S}} = 0.88$), `lbm` ($\hat{\mathcal{S}} = 0.77$ at 500K), and `cactusADM` ($\hat{\mathcal{S}} = 0.90$) also fall in this category. One can visually infer that the shapes of the curves are correlated, as shown in Appendix C. A fewer number of bins can minimize this kind of quantitative comparison error.
- *Model problem.* The histogram conversion algorithm makes no assumption of the conditional probability of

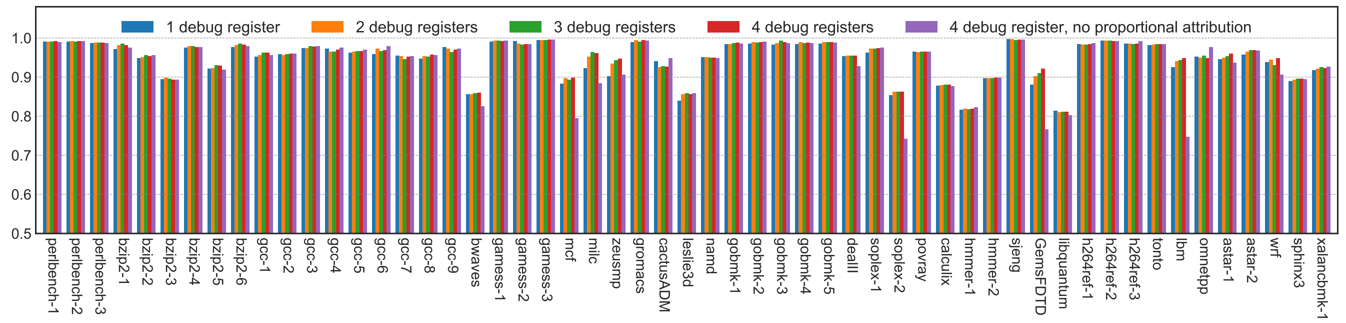


Fig. 6: The accuracy of time reuse distance for all SPEC CPU2006 benchmarks by using different numbers of debug registers or disabling proportional attribution under 5M sampling period.

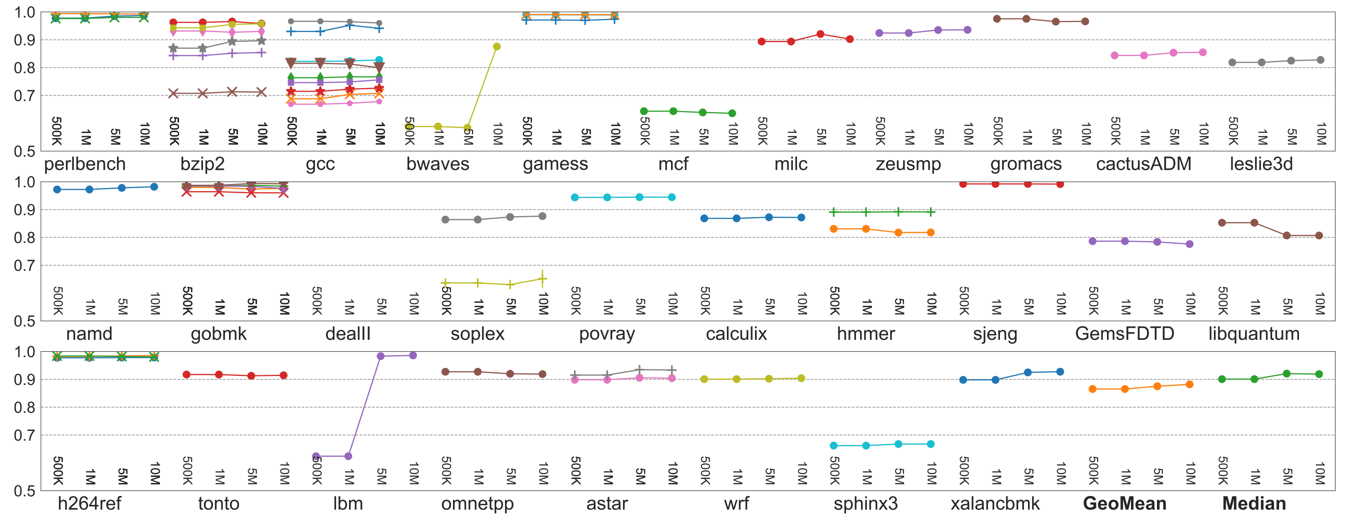


Fig. 7: RDX stack reuse distance accuracy for SPEC CPU2006 benchmarks at 500K, 1M, 5M and 10M sampling periods. One benchmark may have multiple lines due to runs with different inputs. `dealII` does not have the accuracy data because our ground truth tool runs out of memory.

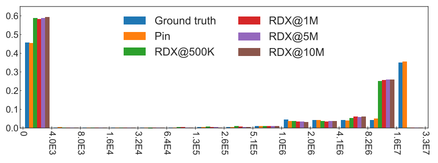


Fig. 8: Stack reuse histogram of `soplex`.

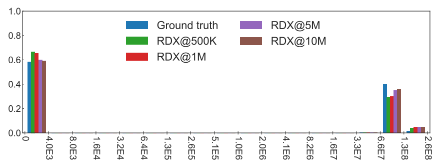


Fig. 9: Time reuse histogram of `lbm`.

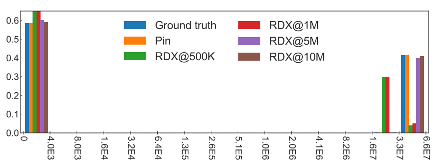


Fig. 10: Stack reuse histogram of `lbm`.

accessing two elements together, which brings some inaccuracy in the model for those benchmarks where accesses are highly dependent on one another. To verify that the inaccuracy comes from the histogram conversion model, we use our Pin-based ground truth tool to collect the *time reuse* histograms for all SPEC CPU2006 benchmarks and covert them to stack reuse histograms. If the Pin-based time distance histograms do not produce highly accurate stack reuse histograms, we classify them into the model problem. `bzip2` (the two lowest accuracy inputs), `gcc`, and `mcf` belong to this category.

- *Sparse reservoir samples.* Sometimes accuracy varies with the sampling period. For example, `bwave` and `lbm`'s accuracy increases with larger sampling period. The sharp accuracy drop in these benchmarks at shorter sampling periods is the cascade of two problems: first, a large number of samples lead to relatively longer survival of old samples in the reservoir, resulting in a mild bias in accounting. Second, the mild bias in reservoir sampling results in off-by-one bin shift for a single bin that contributes to a large fraction of reuse distance. Figs. 9 and 10 respectively show the time reuse and stack reuse histograms for `lbm` demonstrating this effect. `libquantum` suffers from the opposite problem—increasing the sampling period results in sparse PMU samples, but the effect flattens beyond 5M sampling period.
- *PMU imprecision.* Intel precise event-based sampling (PEBS) can suffer from shadow effects [60–62]. PEBS

Table II: The time overhead and memory overheads of SPEC CPU2006 under different sampling periods. If multiple inputs are available for a benchmark, we report the average across all the inputs.

Benchmark			400.perlbench	401.bzip2	403.gcc	410.bwaves	416.gamess	429.mcf	433.milc	434.zeusmp	435.gromacs	436.cactusADM	437.leslie3d	444.namd	445.gobmk	447.dealII	450.soplex	453.povray	454.calculix	456.hmmer	458.sjeng	459.GemsFDTD	462.libquantum	464.h264ref	465.tonto	470.lbm	471.omnetpp	473.astar	481.wrf	482.sphinx3	483.xalancbmk	**GeoMean**	**Median**
Native Run	**Time (s)**		68.71	60.04	24.30	299.24	50.40	215.71	453.70	349.59	308.21	364.60	198.80	298.69	67.28	243.14	85.39	121.54	565.34	149.43	380.76	286.86	198.82	105.13	412.56	354.85	179.25	168.65	291.31	429.86	147.29		
	Memory (MB)		361.65	397.45	382.04	875.53	22.88	1677.43	681.13	514.64	16.43	661.94	125.23	47.92	29.83	796.18	231.74	7.08	118.88	11.79	176.18	830.34	97.70	43.80	38.74	410.52	171.31	210.16	695.78	45.11	422.64		
Supervised Run Overhead @ Different Sampling Periods	**500K**	**Time**	1.36	1.20	1.26	1.16	1.53	1.04	1.05	1.11	1.20	1.19	1.27	1.16	1.30	1.23	1.13	1.34	1.13	1.30	1.17	1.09	1.11	1.35	1.24	1.05	1.15	1.05	1.19	1.14	7.57	1.28	1.24
		Memory	1.04	1.03	1.07	1.01	1.47	1.00	1.02	1.04	1.61	1.01	1.07	1.20	6.57	1.02	1.06	3.67	1.11	1.71	1.17	1.01	1.09	1.32	2.06	1.02	1.06	1.04	1.04	1.22	1.76	1.38	1.07
	1M	**Time**	1.22	1.13	1.19	1.10	1.34	1.01	1.03	1.08	1.12	1.12	1.21	1.09	1.18	1.12	1.08	1.17	1.08	1.17	1.07	1.06	1.08	1.22	1.13	1.04	1.06	1.04	1.12	1.07	4.52	1.18	1.16
		Memory	1.04	1.03	1.06	1.01	1.47	1.00	1.02	1.03	1.62	1.01	1.06	1.21	4.24	1.02	1.07	3.39	1.10	1.67	1.14	1.01	1.07	1.28	1.85	1.02	1.06	1.04	1.04	1.21	1.56	1.31	1.07
	5M	**Time**	1.08	1.07	1.12	1.03	1.18	1.00	1.01	1.05	1.04	1.07	1.10	1.02	1.06	1.04	1.04	1.03	1.04	1.05	0.99	1.03	1.04	1.10	1.04	1.02	1.03	1.04	1.05	1.02	1.81	1.08	1.05
		Memory	1.03	1.03	1.04	1.01	1.43	1.00	1.02	1.02	1.61	1.01	1.07	1.20	2.03	1.02	1.06	2.70	1.08	1.68	1.09	1.01	1.08	1.26	1.51	1.02	1.05	1.04	1.03	1.20	1.33	1.21	1.07
	10M	**Time**	1.06	1.06	1.11	1.03	1.17	1.01	1.01	1.03	1.03	1.05	1.08	1.02	1.05	1.03	1.03	1.00	1.03	1.03	0.99	1.03	1.01	1.08	1.02	1.01	0.99	1.04	1.04	1.00	1.43	1.06	1.04
		Memory	1.03	1.03	1.03	1.01	1.39	1.00	1.01	1.02	1.49	1.01	1.06	1.16	1.69	1.01	1.05	2.35	1.08	1.67	1.08	1.01	1.08	1.24	1.42	1.02	1.05	1.04	1.02	1.22	1.28	1.17	1.06

intends to monitor memory loads or stores with long latency in the pipeline; loads and stores with short latency can hide behind long-latency ones with smaller probability in being sampled. `hmmer` belongs to this category, where we noticed more samples to one instruction compared to another both belonging the same basic block.

We end this subsection by mentioning that despite various causes for accuracy loss, RDX is immune to errors for a vast majority of cases and remains stable across a wide range of sampling periods.

B. RDX Overhead

Table II shows the time and memory overheads at different sampling periods. When the sampling period increases, the overhead also drops. When sampling loads and stores each at 5M, RDX typically incurs 5% runtime overhead and 7% memory overhead. There are some outliers. Benchmarks with deep recursion, e.g., `xalancbmk`, `gamess`, and `gobmk`, incur a higher runtime and memory overhead due to the higher cost of stack unwinding and maintaining their long call paths. Benchmarks with very small memory footprint (e.g., `povray` and `hmmer`) show a higher memory overhead, because of pre-allocated ($\sim$5MB) data structures in RDX. In contrast, `Loca` [33], the state-of-the-art instrumentation-based tool, incurs 153$\times$ runtime slowdown to collect approximated stack reuse histograms. In summary, RDX introduces low time and memory overheads.

VI. Cases Studies

A. SPEC CPU2017

We use RDX to evaluate SPEC CPU2017 benchmarks with reference inputs. An important metric of comparison is how the reuse distance pattern has changed between SPEC CPU2006 and SPEC CPU2017. For each benchmark, we find the smallest reuse distance bin b, such that the cumulative sum over all the bins smaller than b, i.e., $\sum_{i=1}^{b} F(bin_i)$, reaches a given percentile value (p-value). Since bins are associated with distance ranges, we show the ending range of such bin. Table III shows the median and max of these values over all benchmarks in both suites covering different percentiles values (70th-95th). *The median SPEC CPU2017 benchmark's reuse no longer fits in the traditional 32KB L1 cache, whereas it did fit in SPEC CPU2006.* We have also plotted the stack reuse histograms for individual SPEC CPU2017 benchmarks in Appendix D.

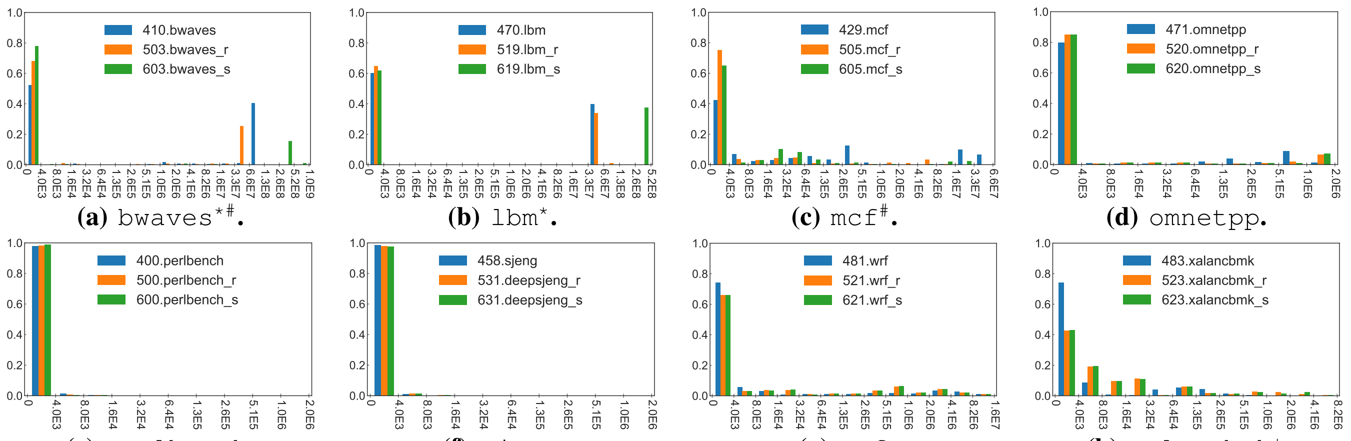
We now focus on a few common benchmarks in two benchmark suites. Each benchmark has three versions: SPEC CPU2006 (`4xx` series), SPEC CPU2017 rate (`5xx` series), and SPEC CPU2017 speed (`6xx` series). Fig. 11 shows a side-by-side comparison of these stack reuse histograms, and we draw the following conclusions.

- *Some benchmarks remain unchanged.* `perlbench`, `wrf`, and `omnetpp` have their histograms unchanged. `perlbench` is CPU bound, with very short reuse distance, almost 100% of reuse distances associated with the smallest bin. `wrf` remains memory bound, the first bin accounts for around 70% of reuse, and the bins from $[4K, 8K)$ to $[8M, 16M)$ have small portions of reuses. `omnetpp`'s histograms are similar to `wrf`.
- *Some benchmarks change significantly.* The evolution of the benchmarks from `400` level to `500` and `600` levels has two trends. (1) The benchmarks have more reuses associated with small bins, meaning less memory bound. `mcf` and `bwaves` belong to this category. (2) The benchmarks have more reuses associated with large bins, meaning more memory bound. `xalancbmk` and `lbm` belong to this category. For `xalancbmk`, both `500` and `600` versions have similar histograms, but with more reuses attributed to large-distance bins than its `400` version. For `lbm`, all the three histograms have three spikes. The `400` and `500` versions have similar histograms: $\sim$60% in the bin $[0, 4K)$ and $\sim$40% in the bin $[32M, 64M)$, while the `600` version has the second spike at $[256M, 512M)$.

`xalancbmk`'s longest reuse distances shifts from 500K to 8.2M, which is a $\sim$15x growth. `lbm`'s longest reuse distances shifts from 66M to 520M, which is a $\sim$10x growth. Such unprecedented growth in reuse distance is

Table III: Max and median reuse distances to achieve different percentiles in SPEC CPU2006 vs. 2017.

	70%		80%		90%		95%	
	Max	Median	Max	Median	Max	Median	Max	Median
SPEC 2006	128MB	4KB	128MB	32KB	128MB	32KB	128MB	32KB
SPEC 2017	512MB	4KB	512MB	4KB	512MB	64KB	512MB	128KB



(a) `bwaves`$^{*\#}$. **(b)** `lbm`*. **(c)** `mcf`$^{\#}$. **(d)** `omnetpp`.

(e) `perlbench`. **(f)** `sjeng`. **(g)** `wrf`. **(h)** `xalancbmk`*.

* significant lengthening of reuse distances in SPEC CPU2017. $^{\#}$ shrinking of reuse distances in SPEC CPU2017.

Fig. 11: Stack reuse distance histograms of common benchmarks in SPEC CPU2006 and CPU2017.

representative of the HPC big-data trends [63] and growth in cloud service workloads, which frequently use data serialization and deserialization between microservices [46, 64, 65]. While past decade has seen unprecedented growth of these kinds of workloads, the hardware trend in last-level cache-sizes does not match the same growth rate, which results in higher latency and reduced IPC for such workloads.

B. *RDX-guided Locality Optimization*

Although this paper primarily focuses on the reuse-distance histograms, RDX has additional capabilities to a) collect PMU metrics, e.g., cache miss numbers and latency above threshold, and b) associate measured use-reuse pairs to application source code with full execution calling contexts. Although the number of memory locations can be large, there are far fewer "hot" code locations involved. Moreover, RDX orders the use-reuse pairs by their frequency occurrence. A large reuse distance and a high memory access latency indicate the symptom of a problem; the code locations of the use-reuse pairs pinpoint the provenance of the problem. The use-reuse context pairs (along with their source code locations) are presented through a GUI with controls to inspect at different abstractions, e.g., module, file, function, loop, and statement. Thus, RDX offers a new locality optimization strategy.

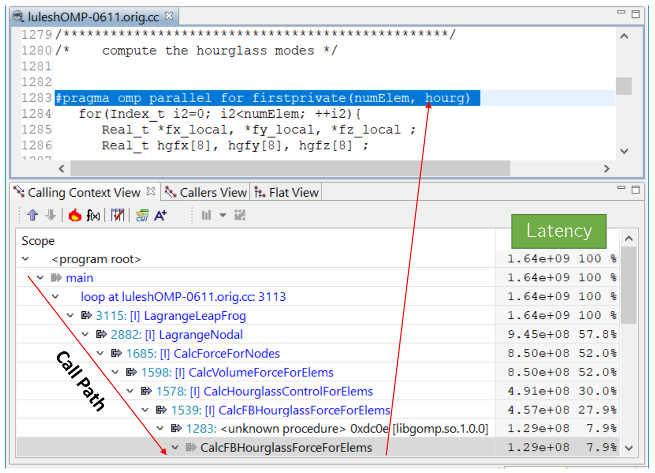
We applied the following strategy to optimize a handful of applications summarized in Table IV. First, locate high-latency memory accesses and identify the corresponding use-and-reuse code locations. Second, filter applications based on their reuse-distance profiles to only focus on the ones with a significant fraction (>50%) of large reuse distances. Third, with the full calling context information, hoist the use and reuse points to the least common ancestor of the two calling contexts via code motion, if safe. We employ loop fusion or tiling to improve temporal locality, and loop interchange or data layout transformation to improve spatial locality. We use LULESH as an example to illustrate the optimization workflow under the guidance of RDX.

LULESH [66], a UHPC application benchmark developed by Lawrence Livermore National Laboratory (LLNL), solves a simple Sedov blast problem with analytical answers. We run it with 32 threads. Fig. 12 shows RDX's graphic user interface highlighting the locations with high latency. The interface consists of three panels. The top one displays the source code; the bottom left one shows the application contexts with functions, loops, and statements; the bottom right panel displays metrics related to the contexts.

RDX shows that a loop (line:1284) accounts for 7.9% of the total latency as shown in Fig. 12. RDX correlates all the reuse pairs with this problematic loop. We find that the application stores the arrays `x8n`, `y8n`, and `z8n` in the loop (line:1509) and later uses them many times in the loop (line:1284) inside the function `CalcFBHourglassForceForElems`. Fig. 13 shows a reuse pair accessing array `z8n`. The time reuse distance is as high as $\sim 4 \times 10^7$ for `x8n`, `y8n`, and `z8n`, indicating poor temporal locality. To enhance the temporal locality, we fuse the two loops (line:1509 and 1284). The fusion is safe: they share the same loop bounds and stride and do not change the dependence direction after the fusion. This fusion optimization yields a $1.54\times$ speedup.

Table IV: Overview of the locality optimization guided by RDX profiles.

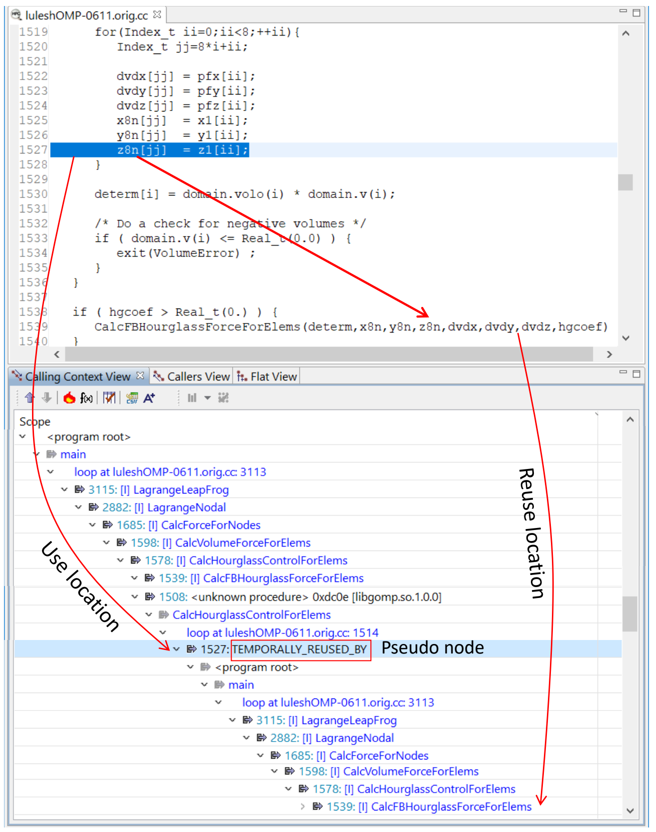
Programs	Locality optimization	Optimization	Speedup
LULESH [66]	temporal locality	fuse loops at line 1509 and 1284 in `lulesh.cc`	1.54×
botsspar [67]	spatial & temporal locality	interchange j-loop and k-loop in function `(bmod) in sparselu.c`	3.45×
backprop [68]	spatial locality	interchange j-loop and k-loop in function `bpnn_adjust_weights` in `backprop.c`	1.52×
srad_v1 [68]	spatial locality	interchange the i-loop and j-loop at line 241 and 242 in `main.c`	1.80×
sweep3d [69]	spatial locality	transpose the last dimension of arrays `flux` and `src` to the second one in `sweep.f`	1.04×



```
1279 /***************************************************/
1280 /*    compute the hourglass modes */
1281
1282
1283 #pragma omp parallel for firstprivate(numElem, hourg)
1284    for(Index_t i2=0; i2<numElem; ++i2){
1285       Real_t *fx_local, *fy_local, *fz_local ;
1286       Real_t hgfx[8], hgfy[8], hgfz[8] ;
```

```
Scope
<program root>                                                    1.64e+09 100 %
  main                                                            1.64e+09 100 %
    loop at luleshOMP-0611.orig.cc: 3113                          1.64e+09 100 %
      3115: [I] LagrangeLeapFrog                                  1.64e+09 100 %
        2882: [I] LagrangeNodal                                   9.45e+08 57.8%
          1685: [I] CalcForceForNodes                             8.50e+08 52.0%
            1598: [I] CalcVolumeForceForElems                     8.50e+08 52.0%
              1578: [I] CalcHourglassControlForElems              4.91e+08 30.0%
                1539: [I] CalcFBHourglassForceForElems            4.57e+08 27.9%
                  1283: <unknown procedure> 0xdc0e [libgomp.so.1.0.0] 1.29e+08  7.9%
                    CalcFBHourglassForceForElems                  1.29e+08  7.9%
```

Fig. 12: Locating a loop region with high latency in LULESH.



```
1519       for(Index_t ii=0;ii<8;++ii){
1520          Index_t jj=8*i+ii;
1521
1522          dvdx[jj] = pfx[ii];
1523          dvdy[jj] = pfy[ii];
1524          dvdz[jj] = pfz[ii];
1525          x8n[jj]  = x1[ii];
1526          y8n[jj]  = y1[ii];
1527          z8n[jj]  = z1[ii];
1528       }
1529
1530       determ[i] = domain.volo(i) * domain.v(i);
1531
1532       /* Do a check for negative volumes */
1533       if ( domain.v(i) <= Real_t(0.0) ) {
1534          exit(VolumeError) ;
1535       }
1536    }
1537
1538    if ( hgcoef > Real_t(0.) ) {
1539       CalcFBHourglassForceForElems(determ,x8n,y8n,z8n,dvdx,dvdy,dvdz,hgcoef)
1540    }
```

```
Scope
<program root>
  main
    loop at luleshOMP-0611.orig.cc: 3113
      3115: [I] LagrangeLeapFrog
        2882: [I] LagrangeNodal
          1685: [I] CalcForceForNodes
            1598: [I] CalcVolumeForceForElems
              1578: [I] CalcHourglassControlForElems
                1539: [I] CalcFBHourglassForceForElems
                  1508: <unknown procedure> 0xdc0e [libgomp.so.1.0.0]
                    CalcHourglassControlForElems
                      loop at luleshOMP-0611.orig.cc: 1514
                        1527: TEMPORALLY_REUSED_BY
                          <program root>
                            main
                              loop at luleshOMP-0611.orig.cc: 3113
                                3115: [I] LagrangeLeapFrog
                                  2882: [I] LagrangeNodal
                                    1685: [I] CalcForceForNodes
                                      1598: [I] CalcVolumeForceForElems
                                        1578: [I] CalcHourglassControlForElems
                                          1539: [I] CalcFBHourglassForceForElems
```

Fig. 13: One reuse pair of `z8n` related to the problematic loop shown in Fig. 12.

VII. Related work

There exist a large body of prior art in measuring program locality. In this section, we focus on the techniques based on reuse distance. We first review the techniques that measure and utilize reuse distances. We then review some schemes that reduce the overhead of reuse distance measurement.

A. *Reuse distance measurement and usage*

Reuse distance, especially its histogram, can be used to characterize both software and hardware. For software, one can use reuse distance to derive miss ratios of both private and shared caches with various cache sizes [7, 8], different cache levels [70], different inputs [39, 71], and different thread numbers [13]. For hardware, one can leverage reuse distance to evaluate the cache design in the multi-core system to reduce cache misses [14–17] or improve power efficiency [27]. These approaches, often, exhaustively instrument memory accesses. RDX is different from these tools since it does not instrument but instead uses lightweight PMU sampling and avails itself of characterizing production software with large inputs.

StatCache [30] and StatStack [72], part of ThreadSpotter performance tool [18], respectively collect time and stack distances. They select less than 100 sampling windows of the program for measuring memory reuses. They instrument memory accesses in these episodes of windows and protect the page enclosing the accessed memory addresses. Once a page fault traps, they single step the execution until the instrumented memory locations are retouched. The biggest difference between these tools and RDX is that they still need memory instrumentation. In the default setting, ThreadSpotter incurs ∼17× runtime overhead on average. Although it can reduce the overhead to ∼64% but with the tradeoff of the extremely sparse sampling windows. In contrast, RDX avoids any instrumentation to the binary and heavyweight single stepping mechanism and hence results in ∼5% runtime overhead—more than 10× improvement. Moreover, windowing schemes are susceptible to missing reuses that are larger than the window size. Fig. 14 compares the accuracy of RDX and ThreadSpotter; with the default setting, ThreadSpotter has lower accuracy than RDX. In certain cases, ThreadSpotter has significantly low accuracy, as low as 37% for `astar`. Fig. 15 shows the histogram of `astar`, where ThreadSpotter incorrectly attributes more on the longer reuses and misses many short-distance reuses.

Orthogonal to measurement tools, there are approaches [12, 73, 74] leveraging static program analysis to obtain reuse distance histograms. These approaches need not run the program and independent from the inputs.

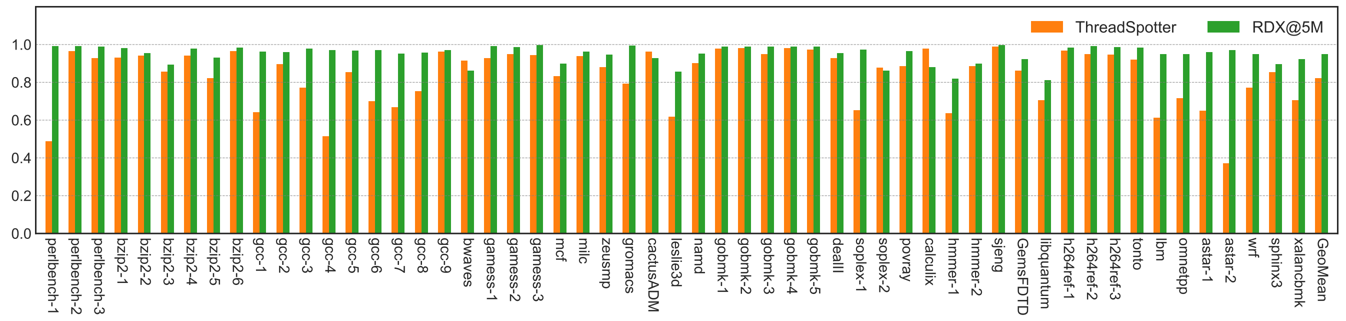


Fig. 14: Time reuse accuracy comparison between ThreadSpotter (running in default sampling rate) and RDX (with 5M sampling period) on SPEC CPU2006 benchmarks. One benchmark may have multiple bars due to runs with different inputs.

However, they are difficult to obtain high accuracy in programs with irregular loops and memory accesses, or intensive usage of pointers and aliases.

B. *Algorithms for accelerating reuse distance measurement*

Collecting thorough reuse distance via exhaustive memory instrumentation incurs more than 100× slowdown [33]. The overhead can easily exceed 1000× when collecting additional information for optimization guidance [21]. Previous studies have devised sampling, approximation, and parallelization strategies to reduce the overhead It is worth noting that these techniques do not abandon instrumentation, so the overhead is still high.

Sampling. Schuff et al. [34] develop a sampling method to allow monitored code running in a fast low-overhead mode, which reduces the measurement overhead to 5.6×. SLO [22] leverages reservoir sampling and reduces the overhead down to 5×. MACPO periodically enables and disables instrumentation to collect reuse information from sampling windows, which reduces overhead to 1.25-5.52×. Zhong et al. [35] invent a sampling algorithm based on the common structure of bursty tracing [75], which incurs 3-10× overhead. Liu and Mellor-Crummey [20] leverage shadow sampling [76] to hide memory instrumentation overhead by forking multiple monitoring processes and overlapping their execution; their tool, however, works on sequential programs only and fails to handle system calls.

Approximation. Ding and Zhong [39] propose a tree compression algorithm to approximate reuse distance. This approach reduces the complexity of tree-based reuse distance measurement from $O(NlogM)$ to $O(NloglogM)$, where N is the length of execution and M is the size of program data. Shen et al. [38] propose statistical models to approximate reuse distance histogram with time distance histogram with accuracy as high as 99% in average. This approximation approach largely reduces the runtime overhead because counting time distance is lightweight without distinguishing unique addresses. RDX also relies on the same model to approximate reuse distance histograms. However, unlike RDX, their approach incurs more than 5× overhead due to the exhaustive memory instrumentation.

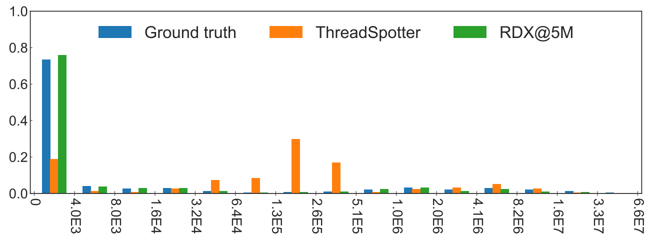


Fig. 15: Time reuse histogram of astar (input 2) from SPEC CPU2006 benchmark suite. ThreadSpotter produces inaccurate results.

Parallelization. To take advantages of computation resources, approaches [36, 37] explore parallel algorithm based on divide-and-conquer to accelerate reuse distance measurement. [36] scales the reuse distance computation to supercomputers, yielding 13-50× speedups compared to the naive method with 64 processors; [37] employs GPUs and incurs 5× slowdown.

VIII. Conclusions

RDX is a lightweight, sampling-based tool for characterizing program data locality via stack reuse distance measurement. RDX distinguishes from prior work by performing no instrumentation. RDX combines the address-sampling capability of hardware performance units with hardware debug registers to sample reuse pairs during program execution. RDX adopts reservoir sampling and proportional attribution to avoid hardware limitations and sampling bias. Evaluated on SPEC CPU2006, RDX yields comparable accuracy as the ground truth, but only incurs 5% runtime and 7% memory overheads. With the help of RDX, we perform the first study of the data locality of SPEC CPU2017 benchmarks and provide the unique insights in both locality characterization and optimization. The low-overhead nature of RDX makes it promising to employ in production environments.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This project was supported by NSF under grant number 1618620.

APPENDIX

A. Time Reuse to Stack Reuse

RDX allocates several bins to include every sampled reuse pair with time distance falling into different ranges. Common to prior work [59, 77], the histogram uses the *logarithmic* scale for the bin size as the horizontal axis and the percentage of total sampled reuse pairs as the vertical axis.

With the time distance histogram, RDX leverages the existing technique [59] to approximate the reuse distance histogram. This section only shows the intuitive idea of this approximation. Considering a memory access sequence `aXXXXa`, the time distance of `a` is 5 while we have no idea how many distinct elements between a. If three time reuses are of distance 1 and one reuses is of distance 5, we definitely know there should be only two distinct elements between `a` by exhaustive searching.

However, this greedy method usually does not work for calculating the reuse distance of a common application, especially directly calculating the reuse distance of the specific reuse instance of an element. Thus we seek to a statistical model to calculate the probability of a specific reuse distance instead of obtaining the reuse distance of a specific reuse instance. Assuming that the probability of a data element is independent from others, whether a data element is accessed in a given time interval Δ is actually a Bernoulli process. Then the probability of having k distinct data elements in this Δ interval is

$$P(k,\Delta) = \binom{N}{k} P_{interval}(\Delta)^k (1 - P_{interval}(\Delta))^{(N-k)} \tag{3}$$

where $P_{interval}(\Delta)$ is the probability for a data element to appear in the interval Δ and N is total number of distinct data elements. To calculate $P_R(k)$ (the probability of having reuse distance k for the entire program), we need to consider all the possibilities of having reuse distance of k from the interval length ranging between 1 and T, where T is the total number of memory accesses. The probability of having time interval Δ can be obtained from the time reuse histogram, denoted as $P_T(\Delta)$. Thus we have

$$P_R(k) = \sum_{\Delta=1}^{T} P(k,\Delta) P_T(\Delta) \tag{4}$$

Since $P_{interval}(\Delta)$ can be derived from $P_T(\Delta)$ and the details can be found in [38], Equation (4) transforms time reuse histogram to stack reuse histogram.

Their later work [59] further developed the algorithm by extending each bar width of both time and stack reuse histograms from 1 to any arbitrary number, which is utilized in our work. Their evaluation shows that this model gives more than 99% accuracy as to cache block granularity.

Benchmark	Input argument
perlbench-1	`-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1`
perlbench-2	`-I./lib diffmail.pl 4 800 10 17 19 300`
perlbench-3	`-I./lib splitmail.pl 1600 12 26 16 4500`
bzip2-1	`input.source 280`
bzip2-2	`chicken.jpg 30`
bzip2-3	`liberty.jpg 30`
bzip2-4	`input.program 280`
bzip2-5	`text.html 280`
bzip2-6	`input.combined 200`
gcc-1	`166.i -o 166.s`
gcc-2	`200.i -o 200.s`
gcc-3	`c-typeck.i -o c-typeck.s`
gcc-4	`cp-decl.i -o cp-decl.s`
gcc-5	`expr.i -o expr.s`
gcc-6	`expr2.i -o expr2.s`
gcc-7	`g23.i -o g23.s`
gcc-8	`s04.i -o s04.s`
gcc-9	`scilab.i -o scilab.s`
gamess-1	`< cytosine.2.config`
gamess-2	`< h2ocu2+.gradient.config`
gamess-3	`< triazolium.config`
gobmk-1	`--quiet --mode gtp" < 13x13.tst`
gobmk-2	`--quiet --mode gtp" < nngs.tst`
gobmk-3	`--quiet --mode gtp" < score2.tst`
gobmk-4	`--quiet --mode gtp" < trevorc.tst`
gobmk-5	`--quiet --mode gtp" < trevord.tst`
soplex-1	`-s1 -e -m45000 pds-50.mps`
soplex-2	`-m3500 ref.mps`
hmmer-1	`nph3.hmm swiss41`
hmmer-2	`--fixed 0 --mean 500 --num 500000 --sd 350 --seed 0 retro.hmm`
h264ref-1	`-d foreman_ref_encoder_baseline.cfg`
h264ref-2	`-d foreman_ref_encoder_main.cfg`
h264ref-3	`-d sss_encoder_main.cfg`
astar-1	`BigLakes2048.cfg`
astar-2	`rivers.cfg`

Fig. 16: The input arguments of benchmarks which have several *ref* inputs for SPEC CPU 2006.

B. Input of SPEC CPU Benchmarks

We use *ref* input to run all SPEC CPU benchmarks but some have multiple inputs, which are distinguished with numerical suffixes such as `gcc-1`, `gcc-2`, etc. The actual mapping of these names are shown in Fig. 16 and Fig. 17.

C. Stack Reuse Histograms of SPEC CPU 2006 Benchmarks with Off-by-one Problem

In SPEC CPU2006, `bwaves` (Fig. 18), `leslie3d` (Fig. 19), `sphinx3` (Fig. 20), `GemsFDTD` (Fig. 21), `lbm` (Fig. 22), and `cactusADM` (Fig. 23) have the off-by-one problem, which can be inferred from their stack reuse histograms.

D. Stack Reuse Histograms of SPEC CPU2017

We have plotted the stack reuse histograms of `perlbench_r` (Fig. 24, Fig. 25 and Fig. 26), `gcc_r` (Fig. 27, Fig. 28, Fig. 29, Fig. 30 and Fig. 31), `bwaves_r` (Fig. 32, Fig. 33, Fig. 34 and Fig. 35), `mcf_r` (Fig. 36), `cactuBSSN_r` (Fig. 37), `namd_r` (Fig. 38), `povray_r` (Fig. 39), `lbm_r` (Fig. 40), `omnetpp_r` (Fig. 41), `wrf_r` (Fig. 42), `xalancbmk_r` (Fig. 43), `x264_r` (Fig. 44, Fig. 45 and Fig. 46), `blender_r` (Fig. 47), `cam4_r` (Fig. 48), `deepsjeng_r` (Fig. 49), `imagick_r` (Fig. 50), `leela_r` (Fig. 51), `nab_r` (Fig. 52), `exchange2_r` (Fig. 53), `fotonik3d_r` (Fig. 54), `roms_r` (Fig. 55), `xz_r` (Fig. 56, Fig. 57 and Fig. 58), `perlbench_s` (Fig. 59, Fig. 60 and Fig. 61), `gcc_s` (Fig. 62, Fig. 63 and Fig. 64), `bwaves_s` (Fig. 65 and Fig. 66), `mcf_s` (Fig. 67), `cactuBSSN_s` (Fig. 68), `lbm_s` (Fig. 69), `omnetpp_s` (Fig. 70), `wrf_s` (Fig. 71), `xalancbmk_s` (Fig. 72), `x264_s` (Fig. 73, Fig. 74 and Fig. 75), `cam4_s` (Fig. 76), `pop2_s` (Fig. 77), `deepsjeng_s` (Fig. 78), `imagick_s` (Fig. 79), `leela_s` (Fig. 80), `nab_s` (Fig. 81), `exchange2_s` (Fig. 82), `fotonik3d_s` (Fig. 83), `roms_s` (Fig. 84), `xz_s` (Fig. 85 and Fig. 86) from SPEC CPU2017.

Benchmark	Input argument
perlbench_r-1	`-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1`
perlbench_r-2	`-I./lib diffmail.pl 4 800 10 17 19 300`
perlbench_r-3	`-I./lib splitmail.pl 6400 12 26 16 100 0`
gcc_r-1	`gcc-pp.c -O3 -finline-limit=0 -fif-conversion -fif-conversion2 -o gcc-pp.opts-O3_-finline-limit_0_-fif-conversion_-fif-conversion2.s`
gcc_r-2	`gcc-pp.c -O2 -finline-limit=36000 -fpic -o gcc-pp.opts-O2_-finline-limit_36000_-fpic.s`
gcc_r-3	`gcc-smaller.c -O3 -fipa-pta -o gcc-smaller.opts-O3_-fipa-pta.s`
gcc_r-4	`ref32.c -O5 -o ref32.opts-O5.s`
gcc_r-5	`ref32.c -O3 -fselective-scheduling -fselective-scheduling2 -o ref32.opts-O3_-fselective-scheduling_-fselective-scheduling2.s`
bwaves_r-1	`bwaves_1 < bwaves_1.in`
bwaves_r-2	`bwaves_2 < bwaves_2.in`
bwaves_r-3	`bwaves_3 < bwaves_3.in`
bwaves_r-4	`bwaves_4 < bwaves_4.in`
x264_r-1	`--pass 1 --stats x264_stats.log --bitrate 1000 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_r-2	`--pass 2 --stats x264_stats.log --bitrate 1000 --dumpyuv 200 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_r-3	`--seek 500 --dumpyuv 200 --frames 1250 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
xz_r-1	`cld.tar.xz 160 19cf30ae51eddcbefda78dd06014b4b96281456e078ca7c13e1c0c9e6aaea8dff3efb4ad6b0456697718cede6bd5454852652806a657bb56e07d61128434b474 59796407 61004416 6`
xz_r-2	`cpu2006docs.tar.xz 250 055ce243071129412e9dd0b3b69a21654033a9b723d874b2015c774fac1553d9713be561ca86f74e4f16f22e664fc17a79f30caa5ad2c04fbc447549c2810fae 23047774 23513385 6e`
xz_r-3	`input.combined.xz 250 a841f68f38572a49d86226b7ff5baeb31bd19dc637a922a972b2e6d1257a890f6a544ecab967c313e370478c74f760eb229d4eef8a8d2836d233d3e9dd1430bf 40401484 41217675 7`
perlbench_s-1	`-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1`
perlbench_s-2	`-I./lib diffmail.pl 4 800 10 17 19 300`
perlbench_s-3	`-I./lib splitmail.pl 6400 12 26 16 100 0`
gcc_s-1	`gcc-pp.c -O5 -fipa-pta -o gcc-pp.opts-O5_-fipa-pta.s`
gcc_s-2	`gcc-pp.c -O5 -finline-limit=1000 -fselective-scheduling -fselective-scheduling2 -o gcc-pp.opts-O5_-finline-limit_1000_-fselective-scheduling_-fselective-scheduling2.s`
gcc_s-3	`gcc-pp.c -O5 -finline-limit=24000 -fgcse -fgcse-las -fgcse-lm -fgcse-sm -o gcc-pp.opts-O5_-finline-limit_24000_-fgcse_-fgcse-las_-fgcse-lm_-fgcse-sm.s`
bwaves_s-1	`bwaves_1 < bwaves_1.in`
bwaves_s-2	`bwaves_2 < bwaves_2.in`
x264_s-1	`--pass 1 --stats x264_stats.log --bitrate 1000 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_s-2	`--pass 2 --stats x264_stats.log --bitrate 1000 --dumpyuv 200 --frames 1000 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
x264_s-3	`--seek 500 --dumpyuv 200 --frames 1250 -o BuckBunny_New.264 BuckBunny.yuv 1280x720`
xz_s-1	`cpu2006docs.tar.xz 6643 055ce243071129412e9dd0b3b69a21654033a9b723d874b2015c774fac1553d9713be561ca86f74e4f16f22e664fc17a79f30caa5ad2c04fbc447549c2810fae 1036078272 1111795472 4`
xz_s-2	`cld.tar.xz 1400 19cf30ae51eddcbefda78dd06014b4b96281456e078ca7c13e1c0c9e6aaea8dff3efb4ad6b0456697718cede6bd5454852652806a657bb56e07d61128434b474 536995164 539938872 8`

Fig. 17: The input arguments of benchmarks which have several *ref* inputs for SPEC CPU 2017.

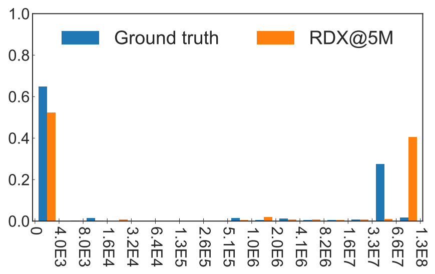


Fig. 18: Stack reuse histogram of `bwaves`.

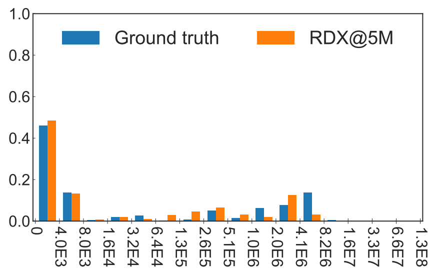


Fig. 19: Stack reuse histogram of `leslie3d`.

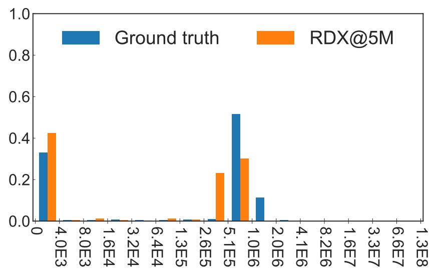


Fig. 20: Stack reuse histogram of `sphinx3`.

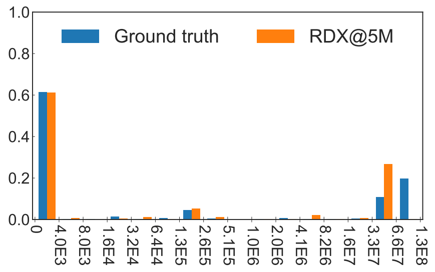


Fig. 21: Stack reuse histogram of `GemsFDTDs`.

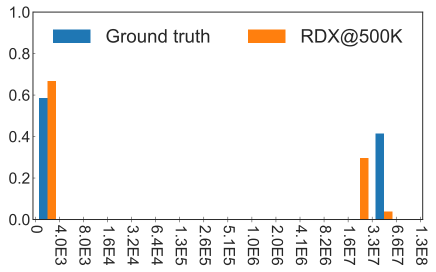


Fig. 22: Stack reuse histogram of `lbm`.

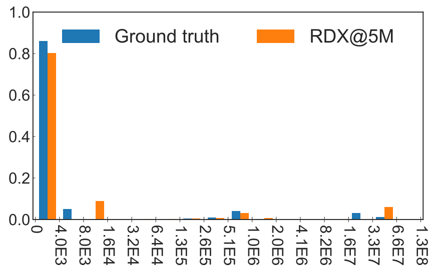


Fig. 23: Stack reuse histogram of `cactusADM`.

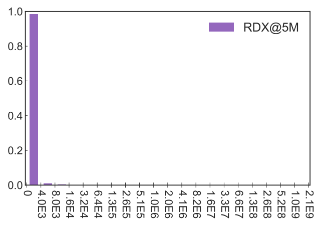


Fig. 24: Stack reuse histogram of `perlbench_r-1`.

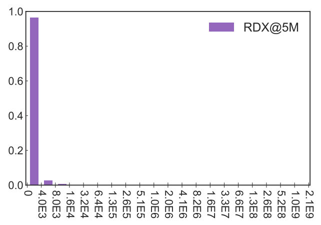


Fig. 25: Stack reuse histogram of `perlbench_r-2`.

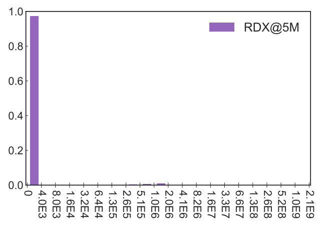


Fig. 26: Stack reuse histogram of `perlbench_r-3`.

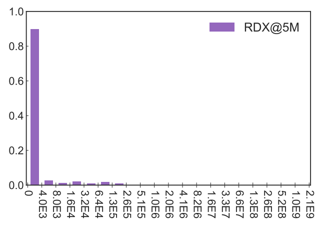


Fig. 27: Stack reuse histogram of `gcc_r-1`.

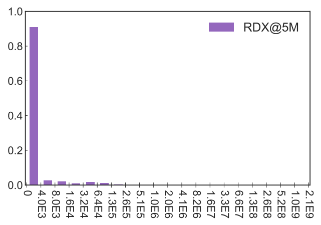


Fig. 28: Stack reuse histogram of `gcc_r-2`.

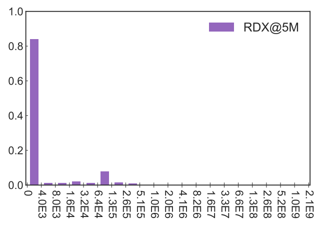


Fig. 29: Stack reuse histogram of `gcc_r-3`.

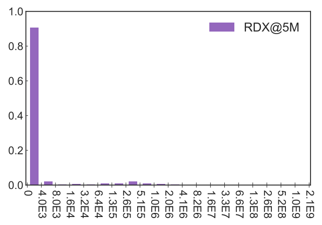


Fig. 30: Stack reuse histogram of `gcc_r-4`.

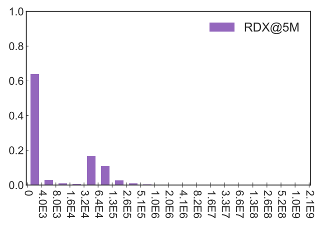


Fig. 31: Stack reuse histogram of `gcc_r-5`.

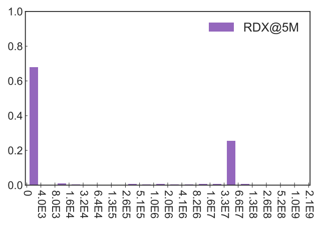


Fig. 32: Stack reuse histogram of `bwaves_r-1`.

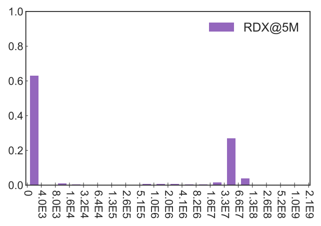


Fig. 33: Stack reuse histogram of `bwaves_r-2`.

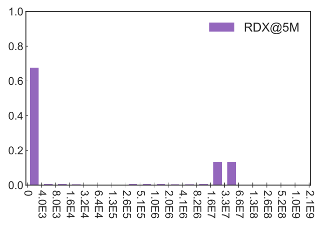


Fig. 34: Stack reuse histogram of `bwaves_r-3`.

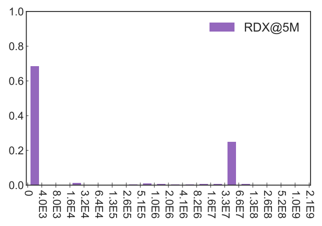


Fig. 35: Stack reuse histogram of `bwaves_r-4`.

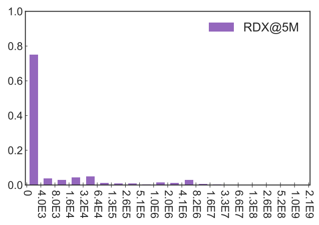


Fig. 36: Stack reuse histogram of `mcf_r`.

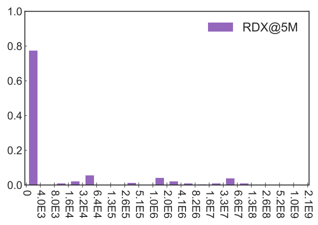


Fig. 37: Stack reuse histogram of `cactuBSSN_r`.

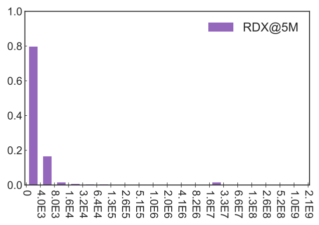


Fig. 38: Stack reuse histogram of `namd_r`.

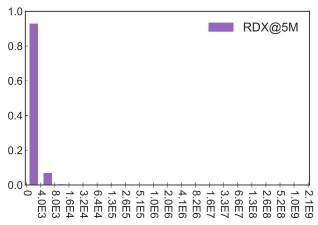


Fig. 39: Stack reuse histogram of `povray_r`.

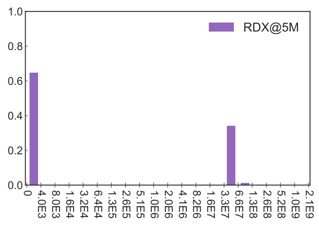


Fig. 40: Stack reuse histogram of `lbm_r`.

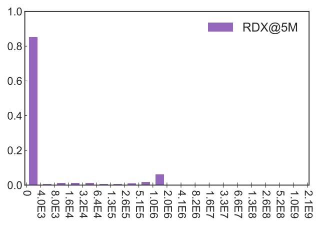


Fig. 41: Stack reuse histogram of `omnetpp_r`.

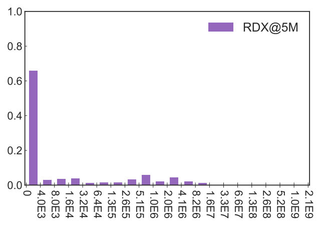


Fig. 42: Stack reuse histogram of `wrf_r`.

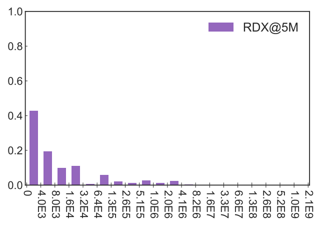


Fig. 43: Stack reuse histogram of `xalancbmk_r`.

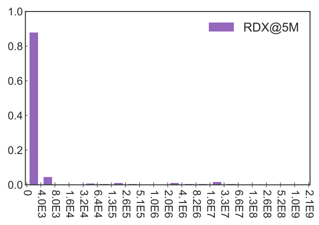


Fig. 44: Stack reuse histogram of `x264_r-1`.

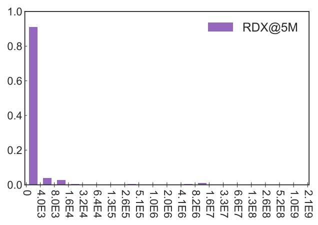


Fig. 45: Stack reuse histogram of `x264_r-2`.

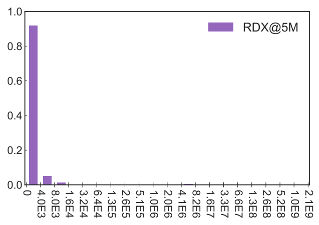


Fig. 46: Stack reuse histogram of `x264_r-3`.

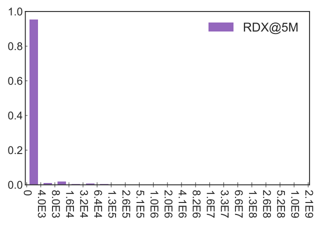


Fig. 47: Stack reuse histogram of `blender_r`.

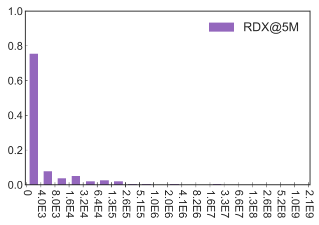


Fig. 48: Stack reuse histogram of `cam4_r`.

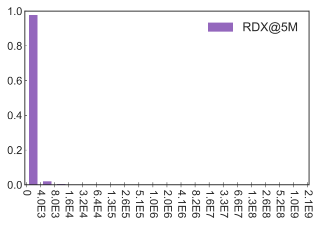


Fig. 49: Stack reuse histogram of `deepsjeng_r`.

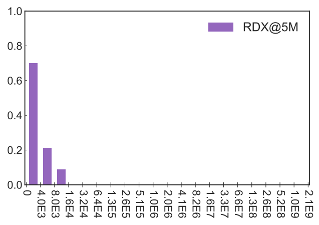


Fig. 50: Stack reuse histogram of `imagick_r`.

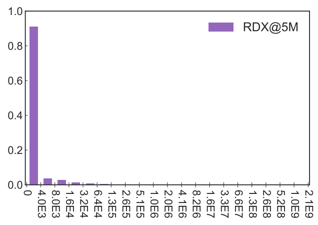


Fig. 51: Stack reuse histogram of `leela_r`.

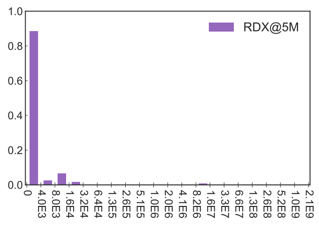


Fig. 52: Stack reuse histogram of `nab_r`.

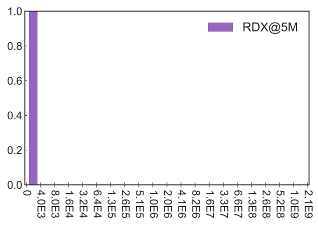


Fig. 53: Stack reuse histogram of `exchange2_r`.

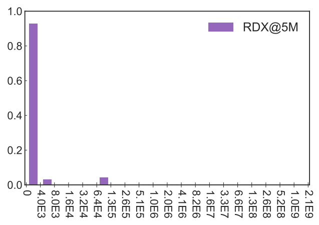


Fig. 54: Stack reuse histogram of `fotonik3d_r`.

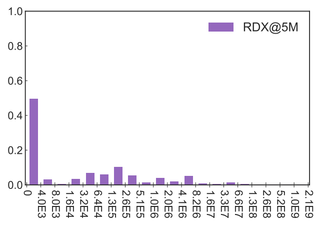


Fig. 55: Stack reuse histogram of `roms_r`.

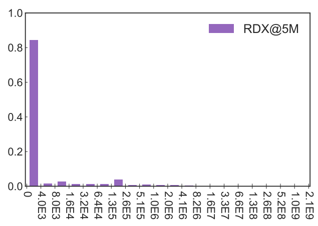


Fig. 56: Stack reuse histogram of `xz_r-1`.

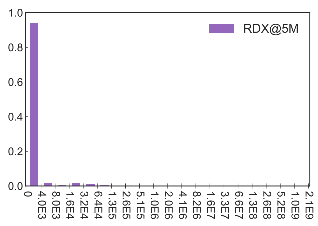


Fig. 57: Stack reuse histogram of `xz_r-2`.

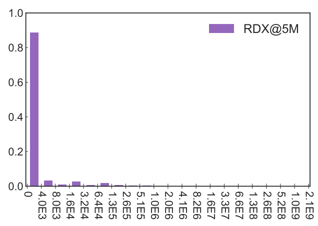


Fig. 58: Stack reuse histogram of `xz_r-3`.

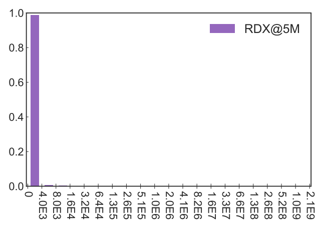


Fig. 59: Stack reuse histogram of `perlbench_s-1`.

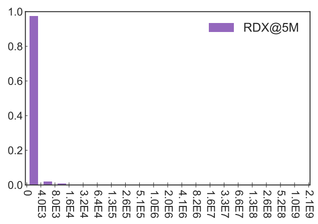


Fig. 60: Stack reuse histogram of `perlbench_s-2`.

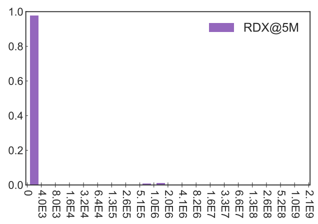


Fig. 61: Stack reuse histogram of `perlbench_s-3`.

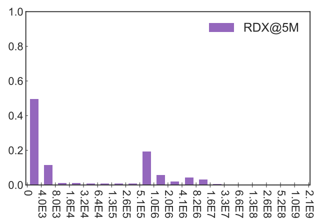


Fig. 62: Stack reuse histogram of `gcc_s-1`.

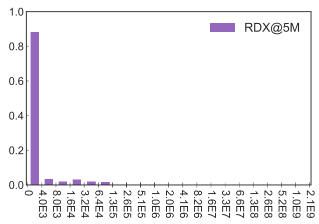


Fig. 63: Stack reuse histogram of `gcc_s-2`.

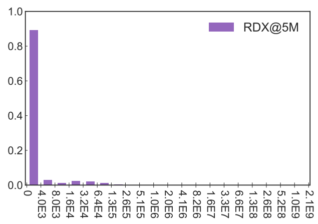


Fig. 64: Stack reuse histogram of `gcc_s-3`.

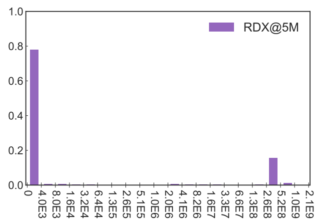


Fig. 65: Stack reuse histogram of `bwaves_s-1`.

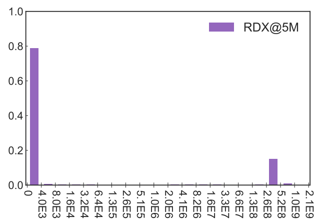


Fig. 66: Stack reuse histogram of `bwaves_s-2`.

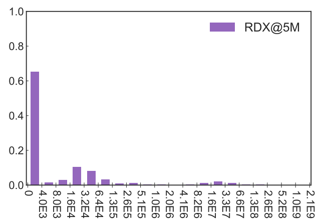


Fig. 67: Stack reuse histogram of `mcf_s`.

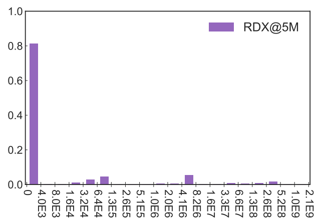


Fig. 68: Stack reuse histogram of `cactuBSSN_s`.

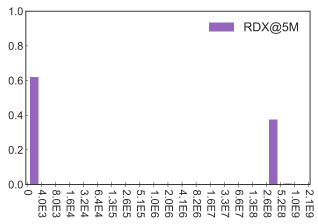


Fig. 69: Stack reuse histogram of `lbm_s`.

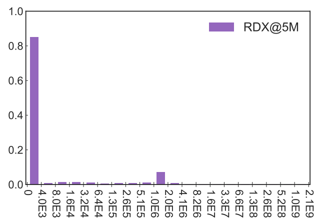


Fig. 70: Stack reuse histogram of `omnetpp_s`.

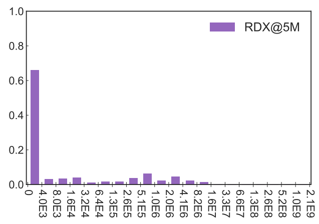


Fig. 71: Stack reuse histogram of `wrf_s`.

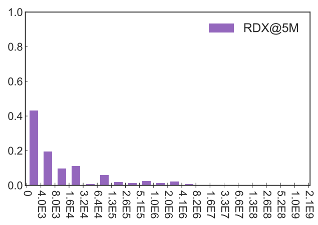


Fig. 72: Stack reuse histogram of `xalancbmk_s`.

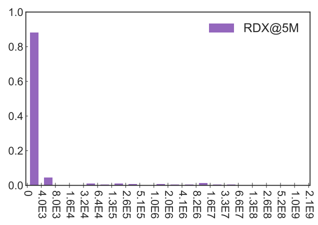


Fig. 73: Stack reuse histogram of `x264_s-1`.

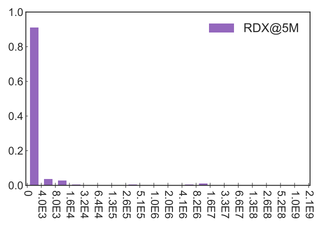


Fig. 74: Stack reuse histogram of `x264_s-2`.

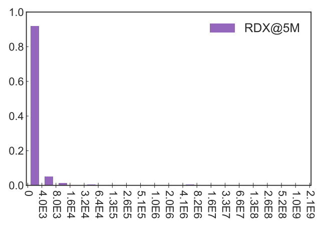


Fig. 75: Stack reuse histogram of `x264_s-3`.

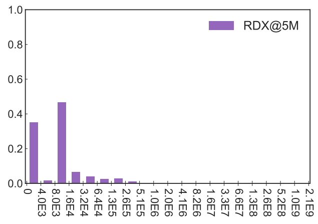


Fig. 76: Stack reuse histogram of `cam4_s`.

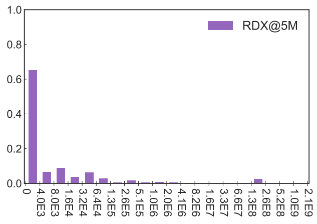


Fig. 77: Stack reuse histogram of `pop2_s`.

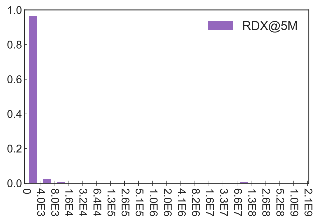


Fig. 78: Stack reuse histogram of `deepsjeng_s`.

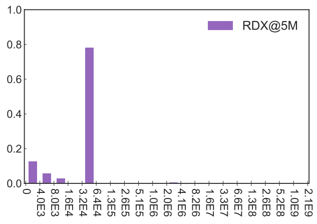


Fig. 79: Stack reuse histogram of `imagick_s`.

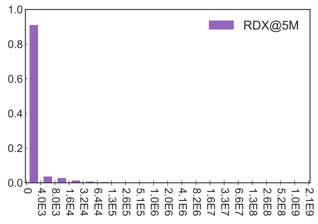


Fig. 80: Stack reuse histogram of `leela_s`.

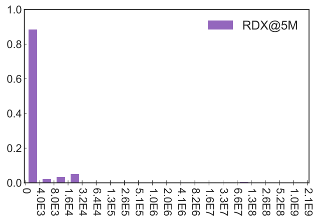


Fig. 81: Stack reuse histogram of `nab_s`.

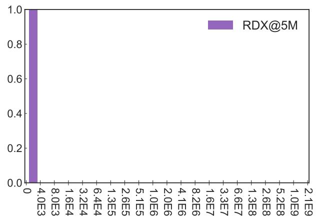


Fig. 82: Stack reuse histogram of `exchange2_s`.

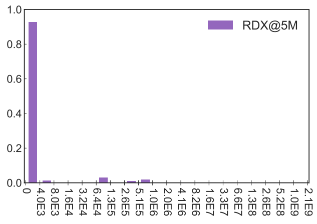


Fig. 83: Stack reuse histogram of `fotonik3d_s`.

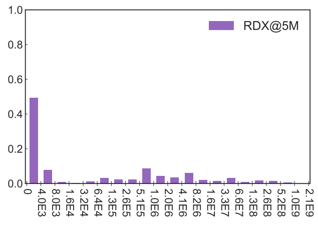


Fig. 84: Stack reuse histogram of `roms_s`.

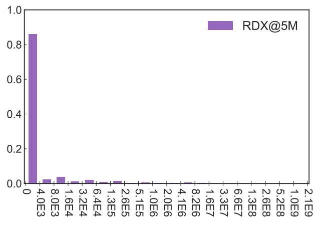


Fig. 85: Stack reuse histogram of `xz_s-1`.

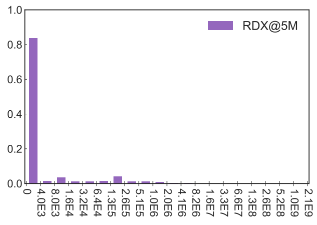


Fig. 86: Stack reuse histogram of `xz_s-2`.

References

[1] IBM Corp., "POWER8 Processor," in *Hot Chips: A Symposium on High Performance Chips*, 2013. [Online]. Available: http://www.hotchips.org

[2] IBM Corp., "POWER9 Processor," https://www.ibm.com/it-infrastructure/power/power9, 2018.

[3] Hewlett Packard Enterprise, "HPE Superdome Flex," https://h20195.www2.hpe.com/v2/GetDocument.aspx?docname=a00026242enw, (Accessed on Dec/12/2018).

[4] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel Xeon Phi processor," in *Hot Chips 27 Symposium (HCS)*. IEEE, 2015.

[5] J. Gecsei, D. R. Slutz, and I. L.Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[6] G. Marin and J. 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*, 2004.

[7] C. Ding and T. Chilimbi, "A composable model for analyzing locality of multi-threaded programs," Tech. Rep., August 2009.

[8] D. Schuff, B. Parsons, and V. S Pai, "Multicore-aware reuse distance analysis," Tech. Rep., 05 2010.

[9] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 37–48.

[10] DynamoRIO team, "drcachesim," http://dynamorio.org/docs/page_drcachesim.html, (Accessed on 03/14/2018).

[11] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI, 2004, pp. 165–176.

[12] K. Beyls and E. H. D'Hollander, "Generating cache hints for improved program efficiency," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 51, no. 4, pp. 223–250, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2004.09.004

[13] M.-J. Wu and D. Yeung, "Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs," *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 264–275, 2011.

[14] M.-J. Wu and D. Yeung, "Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis," in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '12, 2012.

[15] M.-J. Wu and D. Yeung, "Efficient reuse distance analysis of multicore scaling for loop-based parallel programs," *ACM Trans. Comput. Syst.*, vol. 31, no. 1, pp. 1:1–1:37, Feb. 2013.

[16] M.-J. Wu, M. Zhao, and D. Yeung, "Studying multicore processor scaling via reuse distance analysis," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[17] M. Zhao and D. Yeung, "Studying the impact of multicore processor scaling on directory techniques via reuse distance analysis," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 590–602, 2015.

[18] Rogue Wave Software, "ThreadSpotter manual, version 2012.1," http://www.roguewave.com/documents.aspx?Command=Core_Download&EntryId=1492, August 2012.

[19] C. Fang, S. Carr, S. Önder, and Z. Wang, "Reuse-distance-based miss-rate prediction on a per instruction basis," in *Proceedings of the 2004 Workshop on Memory System Performance*, ser. MSP '04, 2004.

[20] X. Liu and J. Mellor-Crummey, "Pinpointing data locality bottlenecks with low overheads," in *Proc. of the 2013 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2013.

[21] K. Beyls and E. H. D'Hollander, "Refactoring for data locality," *Computer*, vol. 42, no. 2, pp. 62 –71, Feb. 2009.

[22] K. Beyls and E. D'Hollander, "Discovery of locality-improving refactorings by reuse path analysis," *Proc. of the* 2^{nd} *Intl. Conf. on High Performance Computing and Communications (HPCC)*, vol. 4208, pp. 220–229, Sept. 2006.

[23] A. Rane and J. Browne, "Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics," in *Proc. of the Intl. Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, USA, 2012.

[24] G. Marin, J. Dongarra, and D. Terpstra, "Miami: A framework for application performance diagnosis," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 158–168.

[25] C. Ding and M. Orlovich, "The potential of computation regrouping for improving locality," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04, 2004.

[26] Y. Zhong, M. Orlovich, X. Shen, and C. 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*, ser. PLDI '04, 2004.

[27] M. Badamo, J. Casarona, M. Zhao, and D. Yeung, "Identifying power-efficient multicore cache hierarchies via reuse distance analysis," *ACM Trans. Comput. Syst.*, vol. 34, no. 1, Apr. 2016.

[28] B. T. Bennett and V. J. Kruskal, "LRU stack processing," *IBM J. Res. Dev.*, vol. 19, no. 4, Jul. 1975.

[29] "dcompiler/loca: Program locality analysis tools," https://github.com/dcompiler/loca, (Accessed on 03/14/2018).

[30] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 169–180, 2005.

[31] G. Marin and J. Mellor-Crummey, "Pinpointing and exploiting opportunities for enhancing data reuse," in *IEEE Intl. Symposium on Performance Analysis of Systems and Software*, 2008.

[32] Y. H. Kim, M. D. Hill, and D. A. Wood, "Implementing stack simulation for highly-associative memories," in *SIGMETRICS*, 1991.

[33] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A higher order theory of locality," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013.

[34] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *Proc. of PACT'10*, 2010.

[35] Y. Zhong and W. Chang, "Sampling-based program locality approximation," in *Proc. of the 7th Intl. Symposium on Memory Management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 91–100.

[36] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "Parda: A fast parallel reuse distance analysis algorithm," in *IEEE 26th Intl.Parallel Distributed Processing Symposium (IPDPS)*, May 2012, pp. 1284–1294.

[37] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng, "A highly parallel reuse distance analysis algorithm on GPUs," in *IEEE 26th Intl. Parallel Distributed Processing Symposium (IPDPS), 2012*.

[38] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," in *Proc. of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, 2007.

[39] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[40] Intel, "Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide," https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf, 2010.

[41] P. J. Drongowski, "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors," https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf, November 2007.

[42] M. Srinivas, B. Sinharoy, R. J. Eickemeyer, R. Raghavan, S. Kunkel, T. Chen, W. Maron, D. Flemming, A. Blanchard, P. Seshadri, J. W. Kellington, A. Mericas, A. E. Petruski, V. R. Indukuru, and S. Reyes, "IBM POWER7 performance modeling, verification, and evaluation," *IBM JRD*, vol. 55, no. 3, pp. 4:1–4:19, May-June 2011.

[43] M. S. Johnson, "Some Requirements for Architectural Support of Software Debugging," in *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[44] R. E. McLear, D. M. Scheibelhut, and E. Tammaru, "Guidelines for Creating a Debuggable Processor," in *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[45] "SPEC CPU2017," https://www.spec.org/cpu2017/, (Accessed on 03/14/2018).

[46] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 158–169. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750392

[47] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did SPEC CPU2017 broaden the performance horizon?" in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[48] Q. Wu, S. Floid, S. Song, J. Deng, and L. K. John, "Hot regions in SPEC CPU2017," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2018.

[49] Linux, "perf_event_open - Linux man page," https://linux.die.net/man/2/perf_event_open, 2012.

[50] J. S. Vitter, "Random Sampling with a Reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, Mar. 1985.

[51] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, 2018.

[52] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency Computation : Practice Expererience*, vol. 22, no. 6, pp. 685–701, Apr. 2010.

[53] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan, "Binary analysis for measurement and attribution of program performance," in *Proceedings of the 2009 ACM PLDI*, NY, NY, USA, 2009.

[54] G. Nakhimovsky, "Debugging and Performance Tuning with Library Interposers," http://dsc.sun.com/solaris/articles/lib_interposers.html, Jul 2001.

[55] Linux, "SIGALTSTACK," http://man7.org/linux/man-pages/man2/sigaltstack.2.html, 2018.

[56] "SPEC CPU2006," https://www.spec.org/cpu2006/.

[57] C.-K. 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," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[58] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, 2007.

[59] X. Shen, J. Shaw, and B. Meeker, "Accurate approximation of locality from time distance histograms," Technical Report TR902, Computer Science Department, University of Rochester, Tech. Rep., 2006.

[60] Levinthal, David, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors, Version 1.0," https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.

[61] A. Nowak, A. Yasin, A. Mendelson, and W. Zwaenepoel, "Establishing a Base of Trust with Performance Counters for Enterprise Workloads," in *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2015.

[62] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for fdo compilation," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[63] M. Asch, T. Moore, R. Badia, M. Beck, P. Beckman, T. Bidot, F. Bodin, F. Cappello, A. Choudhary, B. Supinski, E. Deelman, J. Dongarra, A. Dubey, G. Fox, H. Fu, S. Girona, W. Gropp, M. Heroux, Y. Ishikawa, K. Keahey, D. Keyes, W. Kramer, J. Lavignon, Y. Lu, S. Matsuoka, B. Mohr, D. Reed, S. Requena, J. Saltz, T. Schulthess, R. Stevens, M. Swany, A. Szalay, W. Tang, G. Varoquaux, J. Vilotte, R. Wisniewski, Z. Xu, and I. Zacharov, "Big data and extreme-scale computing: Pathways to convergence-toward a shaping strategy for a future software and data ecosystem for scientific inquiry," in *The International Journal of High Performance Computing Applications*, 2018, pp. 435–479.

[64] D. Namiot and M. sneps sneppe, "On micro-services architecture," in *International Journal of Open Information Technologies*, 2014.

[65] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: Approaches, technologies and research issues," *The VLDB Journal*, vol. 16, no. 3, Jul. 2007.

[66] Lawrence Livermore National Laboratory, "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)," https://computation.llnl.gov/projects/co-design/lulesh.

[67] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux *et al.*, "SPEC OMP2012—an application benchmark suite for parallel systems using OpenMP," in *International Workshop on OpenMP*, 2012.

[68] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[69] A. S. C. Initiative, "The asci sweep3d benchmark code," http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html, 1995.

[70] R. K. V. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and accurate exploration of multi-level caches using hierarchical reuse distance," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 145–156.

[71] Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Transactions on Computers*, vol. 56, no. 3, pp. 328–343, March 2007.

[72] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of LRU caches," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 55–65.

[73] C. Cacaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[74] D. Chen, F. Liu, C. Ding, and S. Pai, "Locality analysis through static parallel sampling," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 557–570. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192402

[75] M. Hirzel and T. Chilimbi, "Bursty tracing: A framework for lowoverhead temporal profiling," in *ACM Workshop on Feedback-Directed and Dynamic Optimization*. ACM, 2001, pp. 117–226.

[76] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri, "Shadow profiling: Hiding instrumentation costs with parallelism," in *International Symposium on Code Generation and Optimization (CGO'07)*, March 2007, pp. 198–208.

[77] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004, pp. 20–27.