REUSETRACKER: Fast Yet Accurate Multicore Reuse Distance Analyzer

MUHAMMAD ADITYA SASONGKO, Koç University, Istanbul, Turkey

MILIND CHABBI, Scalable Machines Research, USA

MANDANA BAGHERI MARZIJARANI and DIDEM UNAT, Koç University, Istanbul, Turkey

One widely used metric that measures data locality is *reuse distance*—the number of unique memory locations that are accessed between two consecutive accesses to a particular memory location. State-of-the-art techniques that measure reuse distance in parallel applications rely on simulators or binary instrumentation tools that incur large performance and memory overheads. Moreover, the existing sampling-based tools are limited to measuring reuse distances of a single thread and discard interactions among threads in multi-threaded programs. In this work, we propose REUSETRACKER—a fast and accurate reuse distance analyzer that leverages existing hardware features in commodity CPUs. REUSETRACKER is designed for multi-threaded programs and takes cache-coherence effects into account. By utilizing hardware features like performance monitoring units and debug registers, REUSETRACKER can accurately profile reuse distance in parallel applications with much lower overheads than existing tools. It introduces only $2.9\times$ runtime and $2.8\times$ memory overheads. Our tool achieves 92% accuracy when verified against a newly developed configurable benchmark that can generate a variety of different reuse distance patterns. We demonstrate the tool's functionality with two use-case scenarios using PARSEC, Rodinia, and Synchronbench benchmark suites where REUSETRACKER guides code refactoring in these benchmarks by detecting spatial reuses in shared caches that are also false sharing and successfully predicts whether some benchmarks in these suites can benefit from adjacent cache line prefetch optimization.

CCS Concepts: • **General and reference** → **Measurement; Performance**; • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: Reuse distance, hardware performance counters, debug registers, address sampling

ACM Reference format:

Muhammad Aditya Sasongko, Milind Chabbi, Mandana Bagheri Marzijarani, and Didem Unat. 2021. REUSETRACKER: Fast Yet Accurate Multicore Reuse Distance Analyzer. *ACM Trans. Arch. Code Optim.* 19, 1, Article 3 (December 2021), 25 pages.

<https://doi.org/10.1145/3484199>

The work is supported by the Scientific and Technological Research Council of Turkey (TUBITAK), Grant No. 120E492. Dr. Didem Unat is supported by the Royal Society-Newton Advanced Fellowship.

Authors' addresses: M. A. Sasongko, M. B. Marzijarani, and D. Unat, Koç University, 34450 Sariyer, Istanbul, Turkey; emails: {msasongko17, mmarzijarani20, dunat}@ku.edu.tr; M. Chabbi, Scalable Machines Research, USA; email: milind@ScalableMachines.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2021/12-ART3 \$15.00

<https://doi.org/10.1145/3484199>

1 INTRODUCTION

Data locality remains an important concern in shared-memory multicore architectures and it is often a more important concern than computation in terms of both energy consumption and performance [54]. Because of its importance, data locality optimizations have become a central focus of application tuning and tool development [55]. One metric widely used to measure data locality is *reuse distance*. Reuse distance is an architecture-independent metric that is defined as the number of unique memory locations that are accessed between two consecutive accesses to a particular memory location (*use* and *reuse*). For example, consider a sequence of accessed memory locations: $a_1, b_1, c_1, b_2, d_1, a_2$. In this example, the reuse distance of a is three, because there are three unique locations accessed between the two consecutive accesses to a , namely, b , c , and d . Reuse distance shows the likelihood of a cache hit for a memory access in a typical **least-recently used (LRU)** cache. If the reuse distance of a memory access is larger than the cache size, then the latter access (*reuse*) is likely to cause a cache miss.

Reuse distance is well-studied for single-threaded programs [6, 8, 10, 15, 21, 28, 37, 49, 64, 65] and a handful of tools are available [1, 60, 64]. The techniques introduced in References [14, 29, 36] employ analytical models that digest and analyze execution traces to predict shared cache behavior. The execution traces needed by these models are generated through *hardware simulation* or *binary instrumentation*. Hardware simulators typically incur significant performance and memory overheads and hence are limited to studying only a small part of a full application. Similarly high overhead exists in binary instrumentation methods. Loca, a reuse distance analysis tool [1, 64] implemented using PIN binary-instrumentation tool [42] incurs 49× runtime and 40× memory overheads, which makes the tool impractical to use in real-life applications. In addition to the overhead problem, authors in Reference [41] show that binary instrumentation can potentially distort the parallel schedule among threads of the profiled program.

Significant overheads and distortion of parallel schedule can also occur in the methods proposed in References [20, 26, 33, 46–48, 59] as they rely on either hardware simulators or binary instrumentation. A technique that avoids these flaws was proposed in Reference [41]. This technique reduces trace size by generating traces at coarser granularity instead of at every load/store access. However, as it operates at coarser granularity, it might miss reuses that are very near in distance while can still capture distant reuses. Another shortcoming of existing reuse distance profilers is that a limited number of them are publicly available. To the best of our knowledge, only Loca [1, 64] and RDX [60] are open source. Yet, both of these tools are limited to measuring reuse distances of a single thread and discard interactions among threads in multi-threaded programs. Considering the wide-spread use of multicore architectures, having a fast yet accurate reuse distance analysis tool for multi-threaded applications is the need of the hour.

This work proposes REUSETRACKER—an open-source, low-overhead reuse distance analysis tool for multi-threaded applications. To make REUSETRACKER fast and memory efficient, we leverage hardware counters of **performance monitoring units (PMUs)** and debug registers, which are commonly available in current commodity CPUs. In REUSETRACKER’s design, PMUs are utilized to periodically sample memory accesses (*uses*) in each profiled thread, while debug registers are used to trap the next accesses (*reuses*) to the sampled memory regions. While the prior PMU-based tools [7, 60] that profile reuse distance have focused only on reuses in a private cache of a core and single-threaded applications, our tool can analyze multi-threaded codes by considering cache line invalidations in detecting reuses in individual threads. Moreover, our tool detects *reuses* across different threads in profiling reuse distance in shared caches. This capability has several applications, one of which is deciding the cache sizes in multicore CPUs.

To account for cache-coherence effects when profiling reuse distance in threads, REUSETRACKER employs a novel algorithm that monitors loads and stores in each thread. A thread that samples

Table 1. Comparison of REUSETRACKER against Other Techniques that Perform Online Detection of *reuses*

Attributes	Schuff et al. [48]	Schuff et al. [47]	StatCache [7]	loca [64]	RDX [60]	ReuseTracker
Time Overhead	High (>100x)	30×	<1.4×	49×	2.03×	2.9×
Space Overhead	Not reported	Not reported	Not reported	40×	2.8×	2.8×
Intra-thread Profiling Accuracy	Close to 100%	96%	*High but not quantified	Close to 100%	90%	92%
Primary Method	Cycle accurate simulator	Binary instrumentation	PMU	Binary instrumentation	PMU	PMU
Profiling Private Cache	YES	YES	YES	YES	YES	YES
Profiling Shared Cache	YES	YES	—	—	—	YES
Cache Line Invalidation	YES	YES	—	—	—	YES
Open Source	—	—	—	YES	YES	YES

Overheads of RDX are measured using a sampling interval of 100K. *StatCache's accuracy is high in predicting miss ratio but it is not quantified in Reference [7].

a particular memory access arms one of its debug registers and the debug registers of other cores to trap any future accesses to the sampled accessed address. When a trap happens, if it pertains to the same core that had sampled the initial access, then reuse is detected. If the trap pertains to a store access in another core, then cache line invalidation is detected, reporting no reuse. When detecting *reuse* in a shared cache, when a trap happens in another core that shares the same cache with the sampling core, a *reuse* is detected. A cache line invalidation is detected when the trap is due to a store access in another core located on a different socket.

Evaluating multithreaded reuse distance, whether for threads or shared caches, turns out to be surprisingly complicated, because one may not know the ground truth reuse distance; the act of measuring the ground truth via instrumentation can perturb the parallel schedule and produce incorrect results unless an additional tool such as Intel's PinPlay [40] that can record and replay application execution deterministically is used. Hence, we developed a benchmark that can generate a variety of reuse distance histograms to aid in validating REUSETRACKER. Our set of benchmarks are useful not only for evaluating this work but can also serve to evaluate future tools in this area of research.

Table 1 compares REUSETRACKER against five similar tools that perform online reuse detection. REUSETRACKER—to the best of our knowledge—is the first tool that accurately profiles reuse distances in threads and shared caches for multi-threaded applications while introducing significantly lower overhead compared to instrumentation-based or simulation-based tools. More specifically, it introduces 2.9× performance and 2.8× memory overheads. REUSETRACKER relies on existing hardware features in commodity CPUs and works on fully optimized binaries, which makes it appropriate for monitoring full applications in production. Furthermore, REUSETRACKER is more accurate compared to other available tools as it profiles multi-threaded code based on real hardware events without dilating the original execution. It achieves 92% accuracy when verified against a configurable-synthetic benchmark that we developed.

In our experimental study, we also demonstrate several use cases of REUSETRACKER. We show that the reuse distance profiles produced by REUSETRACKER can help guide code refactoring to reduce false sharing. Through this code refactoring, we could improve the performance of the benchmarks by up to 87%. Furthermore, we also show that our tool can help predict whether profiled applications can gain or lose performance when **adjacent cache line prefetch (ACP)** is enabled.

In summary, our contributions are listed as follows:

- Formal definitions of reuse in threads and shared caches as *microarchitecture-independent* events in multi-core systems.
- Low-overhead reuse distance analysis algorithms that profile reuse distance in threads and shared caches for multi-threaded programs.
- A synthetic benchmark that can be configured to produce a variety of reuse distance histograms.
- Open source implementation of aforementioned techniques in a tool called REUSETRACKER, as well as extensive evaluation of this tool using a number of benchmarks and applications.
- Demonstration of use-case scenarios using PARSEC [9], Rodinia [12], and Synchrobench [22] benchmark suites where REUSETRACKER guides code refactoring in these benchmarks by detecting spatial reuses in shared cache that are also false sharing and predicts whether some benchmarks in these suites can benefit from ACP.

The repository of REUSETRACKER is publicly available at <https://github.com/ParCoreLab/ReuseTracker>.

2 BACKGROUND AND TERMINOLOGY

In this section, we first provide the background on reuse in single-threaded applications; we then expand the definition to multi-threaded applications; we finally provide a background of a few basic building-block hardware and software facilities needed to develop our tool.

2.1 Single-threaded Reuse Distance

Time Reuse Distance and Reuse Distance: *Time reuse distance* is defined as the number of memory accesses between the current access (*reuse*) and the previous access to the same element (*use*). If the accessed element is in a unit of cache line, then *reuse* can be classified into two, *temporal reuse* and *spatial reuse*. A temporal reuse happens when both *use* and *reuse* access exactly the same address, while a spatial reuse occurs when its *use* and *reuse* access different addresses that are located in the same cache line.

The term *reuse distance* used in this work refers to *LRU stack distance* as defined in Reference [37]. The difference between *time reuse distance* and *reuse distance* is described in the following example. Consider the sequence of memory accesses: $a_1, b_1, c_1, b_2, d_1, d_2, a_2$. In this example, the time reuse distance of a is five, because there are five *total* memory accesses between the first access to a (a_1) and the second access to a (a_2). However, the reuse distance of a is three, since there are only three *unique* accesses between a_1 and a_2 , i.e., b, c , and d .

Time reuse histograms can be accurately translated into stack reuse histograms [50, 60]. REUSETRACKER captures time-reuse during its online profiling and exploits the same conversion techniques during a post-processing step to finally present the stack reuse histograms.

2.2 Multi-threaded Reuse Distance

We now define reuse as a *microarchitecture-independent* event in the context of cache-coherent multi-core systems. For the ease of prose, in the rest of this article, we treat OS threads being pinned to CPU cores and cores are not oversubscribed, hence we use the terms “threads” and “cores” interchangeably.

Definition 2.1 (Intra-thread Reuse). If two consecutive accesses (loads or stores) happen to the same memory location in the same thread without an intervening store to the same location in another thread, then the access pair represents an instance of reuse.

Table 2. Intra-thread Reuse

access order	same core	different cores (any socket)
$R \rightarrow R$	reuse	no reuse
$R \rightarrow W$	reuse	invalidation
$W \rightarrow W$	reuse	invalidation
$W \rightarrow R$	reuse	no reuse

Read (R) or Write (W) may be accessing data at any level in the memory hierarchy.

Table 3. Reuse in Shared Cache

	same core	different cores (same socket)	different cores (different socket)
$R \rightarrow R$	no reuse	reuse	no reuse
$R \rightarrow W$	no reuse	reuse	invalidation
$W \rightarrow W$	no reuse	reuse	invalidation
$W \rightarrow R$	no reuse	reuse	no reuse

The time reuse distance is the number of memory accesses elapsed on the *same thread* between the use and reuse events. As we define intra-thread reuse as a microarchitecture-independent event, we disregard microarchitectural configurations such as private cache size, shared cache size, and cache inclusion policy. Therefore a *reuse* in a thread, no matter how long it is from its *use*, is always counted as a *reuse* even if it hits in a shared cache or DRAM memory in a real machine provided that it still happens in the same thread as the *use*.

Definition 2.2 (Invalidation for Intra-thread Reuse). If two consecutive memory accesses happen to the same memory location by two different threads where the second access is a store operation, then the second access represents an instance of invalidation with respect to the first access.

An invalidation makes the previous access ineligible for reuse with the immediately next access to the same location. Reuse and invalidation are mutually exclusive events. A pair of consecutive memory accesses to the same location can also neither be a reuse nor be an invalidation; for example, a read by one core followed by a read of the same address by another core is neither a reuse nor an invalidation.

Table 2 summarizes reuse, no-reuse, and invalidation in all possible read/write access and same/different core scenarios. $R \rightarrow W$ indicates that a write access happens following the read access to the same location. The Read or Write may be accessing data at any level in the memory hierarchy.

Definition 2.3 (Reuse in Shared Cache). If two consecutive memory accesses happen to the same memory location from two different cores that share the same cache on a multi-core machine, then the access pair represents an instance of reuse in the shared cache.

Definition 2.3 covers a situation where one thread fetched data into the shared cache of a multi-core system and a subsequent access by another core can reuse the same data without having to refetch it from the main memory. We are aware of the possibility that an intra-thread reuse might also hit in a shared cache or DRAM. However, we omit that situation from this definition as it has been covered by our definition of intra-thread reuse, and thus, will be detected by the intra-thread profiling. On multi-socket systems, cores span beyond a single socket. In such cases, writes happening on another socket to the same address can create an invalidation at shared cache level. This is analogous to write accesses on another thread in the intra-thread reuse.

Definition 2.4 (Invalidation in Shared Cache). If two consecutive memory accesses happen to the same memory location by cores on different sockets, where the second access is a store operation, then the second access represents an instance of shared cache invalidation with respect to the first access.

Table 3 summarizes reuse, no-reuse, and invalidation in all possible read/write access and same/different core scenarios. We describe a few cells in more detail.

$R \rightarrow R$: The first access is a read that may access data at any level in the memory hierarchy. (a) If the subsequent read happens on the same core, then it is already captured as a reuse in the intra-thread profiling and thus not a case of shared-cache reuse. (b) If the subsequent read happens on another core of the same socket, then it offers an opportunity to exploit the data fetched by one core into the shared cache to be reused by another core (c) Finally, if the subsequent read happens on another socket, there is no reuse opportunity but it does not invalidate the data on the first socket.¹

$R \rightarrow W$: This is similar to $R \rightarrow R$ except that the second access is a write and the write on a different socket causes an invalidation.

$W \rightarrow W$: The first access is a write. (a) If the subsequent write happens on the same core, then it is already captured as a reuse in the intra-thread profiling and hence it is not counted as a case of shared-cache reuse. (b) If a subsequent write happens on another core of the same socket, then it is guaranteed to miss in its private cache; as before, it offers an opportunity for data reuse at the shared-cache level. (c) Finally, if the subsequent write happens on another socket, it will invalidate the data on the first socket.

$W \rightarrow R$: This is similar to $W \rightarrow W$ except that the subsequent read on another socket does not cause an invalidation on the first socket. A following R/W on the same socket can create a reuse pair.

Since we treat *reuses* and *invalidations* as microarchitecture-independent events, we do not consider microarchitectural configurations such as cache inclusion policy, private cache size, or shared cache size in our definition. Therefore, any memory access that we count as a reuse in shared cache does not necessarily hit in a shared cache in an actual machine. Nevertheless, we can still derive metrics such as L2 or L3 cache miss rates from the reuse distance profiles that follow this definition if the microarchitectural configurations of the machines are known. The reuse distance profiles based on this definition can also recommend suitable L3 cache sizes that optimize the applications' runtime.

Duality property: Shared and intra-thread reuses are the dual of one another. A pair of accesses can either be a reuse in a single thread or in the shared cache, but not both. If an access pair in the same core is counted toward intra-thread reuse, then it is not counted toward shared-cache reuse. If an access pair in the same socket is not counted toward intra-thread reuse, then it is counted toward shared-cache reuse on the same socket. A subsequent remote write operation always invalidates both intra-thread and shared cache reuses. A subsequent remote read operation never causes an invalidation but does not contribute toward reuses neither in a thread nor in a shared cache.

From the hardware perspective, reuse happens at the cache line granularity or sometimes larger than cache line granularities (e.g., Adjacent Cache-Line Prefetch [23]).

2.3 Background on Building Blocks

Hardware Performance Monitoring Unit (PMU): CPU's PMU has the ability to count hardware events such as retired instructions, loads, and stores to name a few. PMUs can be programmed to cause a counter overflow interrupt once a number of events elapse. This number of events is a configurable parameter called *sampling period*. A signal handler, which is a part of REUSETRACKER, handles the OS signal triggered by the interrupt and collects the data, such as accessed effective address, related to the event that causes the interrupt. We refer to a PMU counter overflow interrupt as a "sample." PMUs exist in each CPU core and virtualized by the OS for each OS thread.

¹We note that a different thread pinning can expose the reuse opportunity, but we consider it orthogonal to our current work.

Intel's **Precise Event-based Sampling (PEBS)** [25] allows the programmer to obtain the effective address accessed by the sampled instruction for memory access events such as loads, stores, and cache misses. This address sampling ability is provided in AMD processors by **Instruction-based Sampling (IBS)** feature [17] since AMD Family 10h Processors, in POWER processors by Marked Events feature [52] since POWER 5, and in Intel processors by PEBS since Intel Nehalem and their successors.

Hardware debug registers: Hardware debug registers [30, 38] trap CPU execution when a program counter reaches an instruction address (breakpoint) or when an instruction accesses a monitored address (watchpoint). When a thread sets up a debug register to trap a future memory access to certain address, we consider this thread to be *arming a watchpoint* on that address. One can configure debug registers with different addresses, widths, and types of memory accesses to be trapped (i.e., WP_WRITE for stores and WP_RW for both loads and stores). Current processors from x86 architectures typically have four debug registers.

Linux perf_events: Linux provides an interface to configure PMUs and debug registers using the `perf_event_open` [35] system call and `ioctl` calls. The capability to configure debug registers has been available since Linux 2.6.33, and the capability to program multiple PMUs since Linux 2.6.39 [35]. The Linux kernel can deliver a signal to the specific thread that encounters a PMU interrupt or a debug register trap. The user code can (1) create a circular buffer into which the kernel appends the sampled data on each sample using `mmap` function and (2) collect the signal context on each watchpoint trap.

3 METHODOLOGY

In this section, we provide a high-level sketch of our approach to measuring reuse distance in multi-threaded applications.

Our tool—REUSETRACKER—adopts a sampling philosophy. One need not observe every instance of reuse or invalidation but only a few events in an unbiased manner to produce reuse histograms.² We rely on the statistical significance of reuse or invalidation events to guide us in detecting reuse and invalidation and eventually in computing multi-core reuse distance.

To accomplish the sampling-based monitoring, we need two building block components:

- (1) The ability to sample memory addresses accessed by each thread in the monitored program.
- (2) The ability to identify whether the immediate next memory access to a sampled address happens on the same thread (reuse) or on a different thread (invalidation if the access is a memory write).

In other words, we need the ability to sample pairs of consecutive accesses to the same memory location, irrespective of which thread performs those accesses.

We achieve (1) by using hardware performance monitoring units. As previously described, when programming in sampling mode, on reaching a threshold number of events, i.e., *sampling period*, the PMU can pause the CPU and deliver an interrupt, which can include a packet of information including the memory address being accessed by the thread at the time of counter overflow. This sample is treated as a *use* whose *reuse* or *invalidation* is to be detected later.

We achieve (2) by using hardware debug registers to trap execution when a thread accesses a designated memory address. A sampling thread can arm a watchpoint for itself and for other threads in the process so that any thread accessing a designated address will cause a trap; based on whether the same thread traps or a different thread traps (on a different core or a different socket), we can conclude reuse vs. invalidation. For example, if a watchpoint traps in the sampling

²The same philosophy has been at the core of RDX [60], but it is limited to single-threaded reuse distance measurement.

Fig. 1. One possible execution scenario when profiling intra-thread reuse distance: (1) Every thread sets its PMUs to sample its stores and loads. (2) Thread T_1 's PMU counter overflows on a store to address m_1 . (3) T_1 arms its watchpoint with type WP_RW and watchpoints of other threads (e.g., the one in T_2) with type WP_WRITE and with address m_1 in debug registers. (4) T_1 accesses address m_1 again before any other thread, the watchpoint traps, time reuse distance is computed. (5) Cache line invalidation happens if T_2 stores to address m_1 before T_1 accesses m_1 .

thread, before it traps in any other thread, then an intra-thread reuse is detected and the *time reuse distance* between the *use* and *reuse* is the total loads and stores elapsed on the same thread; if a watchpoint traps in a different thread on the same socket, then a shared cache reuse is detected and the *time reuse distance* between the *use* and *reuse* is the total loads and stores elapsed between the two events on all threads on the same socket.

In practice, one may not be able to sample an address and instantaneously convey the sampled address to all other threads so that they can monitor their access to it. Hence, we mildly relax the definitions as follows.

Relaxed Definition of Reuse. Rather than holding up to the ground truth definition of reuse in Definition 2.1 (2.3), we relax it to mean that *no intervening store happened on another core (another socket) between two consecutively observed accesses to the same memory location by the same thread*. This relaxation accommodates the possibility of the profiler missing an invalidation event in a short window between when the tool samples an address accessed in one thread and informs all other threads to monitor their own accesses to the same address. Thus, if an instance of reuse is detected in this scheme, then there is a high probability that a reuse happened, but we could have missed an invalidation with a small probability.

We do not relax the definition of invalidation, because a thread that produces an address sample can be immediately paused before it can execute the next instruction, thanks to the precise performance events in modern PMUs. Hence, any invalidation detected by a sampling-based profiler is a true invalidation event.

4 DESIGN AND IMPLEMENTATION

REUSETRACKER utilizes two slightly different algorithms to profile reuse distance in individual threads and in shared caches. Next, we present the intra-thread profiling algorithm followed by the one for shared caches.

4.1 INTRA-THREAD PROFILING

Figure 1 shows the main components of the INTRA-THREAD PROFILING algorithm and one of the possible execution scenarios. In this algorithm, REUSETRACKER samples memory store and load events. It handles PMU samples of these events and watchpoint traps with algorithms shown in Algorithms 1 and 2, respectively. In both algorithms, there is a common global data structure,

GlobalWP, which is an array of a struct of length equal to the number of available debug registers per core (e.g., four on an x86 machine). The i th element in GlobalWP holds the information about the watchpoints currently set in the i th debug register of every thread in the monitored process.

ALGORITHM 1: PMU SAMPLE HANDLER

```

1: global GlobalWP[NUM_DEBUG_REGS]
2:
3: procedure PMUSAMPLEHANDLER(Address  $M$ , ThreadID  $T$ ,
  PMUCounterValue  $P$ )
4:   for each slot  $\in \{ \text{GlobalWP} \}$  ordered-by inactive to ac-
     tive do
5:     if AllowReplacement(slot) then
6:       slot.active = true
7:       for each  $T_i \in \{ \text{all threads in the program} \}$  do
8:         if  $T_i == T$  then
9:           mode = WP_RW
10:        else
11:          mode = WP_WRITE
12:        end if
13:        ArmWatchpoint ( $M$ ,  $T_i$ ,  $P$ , slot, mode) re-
14:      turn
15:    end for
16:  end if
17: end procedure
18:
19: procedure ALLOWREPLACEMENT(WPInfo slot)
20:    $r = \text{GenerateRandomNumber}(0.0, 1.0)$ 
21:    $\text{ret} = (r \leq 1.0 / \text{slot.samples}) ? \text{true} : \text{false}$ 
22:   slot.samples++
23:   return ret
24: end procedure
25:
26: procedure ARMWATCHPOINT(Address  $M$ , ThreadID  $T$ ,
  PMUCounterValue  $P$ , WPInfo slot, WPMode  $t$ )
27:   WP.address =  $M$ 
28:   WP.PMUValue =  $P$ 
29:   WP.mode =  $t$ 
30:   Set a WP in the debug register watchpoint slot of thread
   $T$ 
31: end procedure

```

ALGORITHM 2: WATCHPOINT TRAP HANDLER

```

1: procedure WPTRAPHANDLER(ThreadID  $T$ , PMUCounterValue  $P$ ,
  WPInfo slot)
2:
3:   if GlobalWP[slot].active then  $\triangleright$  Check if no thread has
     trapped on this address
4:     if  $T == \text{GlobalWP[slot].ownerTID}$  then
5:        $P_0 = \text{GlobalWP[slot].GetPMUCounterValue}()$ 
6:       Record time reuse distance  $P - P_0$   $\triangleright$  Reuse distance is
         recorded
7:     else
8:       Record invalidation  $\triangleright$  WP type must be WP_WRITE
9:     end if
10:    GlobalWP[slot].active = false
11:    GlobalWP[slot].samples = 1.0
12:  end if
13: end procedure

```

On a PMU Sample: On reaching a threshold number of loads or stores an interrupt is delivered to the thread reaching the threshold. Assume M_1 is the address accessed at the time of interrupt by thread, say T_1 . This signal-handling function, PMUSampleHandler shown in Algorithm 1, takes the sampled address, sampling thread ID, and the total value of load and store PMU counters at the time of sample as inputs. We iterate over all watchpoint slots looking for an available slot (line 4); we look for the unused slots first; the search ends if we find one. If we do not find an unused (inactive) slot, then the search continues into in-use (active) slots. The check AllowReplacement on line 5 is the reservoir sampling logic, which we explain later. If the slot is in-use, then it returns true or false probabilistically. If the slot is inactive, then AllowReplacement will return true, because slot.samples is reset to 1.0 when a watchpoint traps in Line 11 of Algorithm 2. In case of a successful search, the address will be armed on all watchpoints on all cores. However, the search may end without arming any watchpoint, in which case the sampled address will be dropped.

Special attention needs to be placed on the *mode* in which the watchpoints are set up. A subsequent *load* or *store* access to the sampled address by *the same thread*, indicates reuse (see Table 2); hence, the sampling thread, T_1 , arms the watchpoints in its own debug registers in read-write WP_RW mode. A subsequent *store* to the sampled address by *another thread*, indicates invalidation;

hence, the sampling thread arms the watchpoints in debug registers of other threads in write-only WP_WRITE mode.

On watchpoint trap: When a watchpoint traps in a thread, say T_2 , due to a memory access to an address, say M_1 , REUSETRACKER handles the trap with an algorithm shown in Algorithm 2. In Line 3, T_2 checks if the watchpoint *slot* that corresponds to the trapping debug register is still *active* (meaning never trapped in any threads). If it is still active, then T_2 checks if the thread ID of the watchpoint *slot* matches its own thread ID. If it matches, then time reuse distance is computed as *use* and *reuse* occur on the same thread. In this case, the time reuse distance is the number of loads and stores elapsed from the *use* to the *reuse* point, which is read from PMU counters. If the two accesses are from two different threads (the latter one guaranteed to be a write access), then an invalidation is detected. After detecting either a *reuse* or an invalidation, in Line 10, T_2 marks the watchpoint *slot* as *inactive*; subsequent traps, if any, on the same slot will be ignored so that at most one *reuse* or invalidation is detected per sampled address.

After detecting reuse or invalidation, the watchpoint slot is immediately released for use by a subsequent sample. As a special case, if every armed watchpoint traps before the next PMU address sample, then the reservoir sampling ensures all sampled addresses to be monitored.

Reservoir Sampling Strategy: On a PMU sample, a previously armed watchpoint may have already trapped (either because of reuse or invalidation) or not yet trapped. REUSETRACKER needs to balance between retaining a previously armed watchpoint to potentially detect a reuse separated by some other memory accesses and a new address that may trap sooner; neither of these is predictable without time traveling into the future of execution.

We employ reservoir sampling to give equal probability to retain an already monitored address for a longer time vs. beginning to monitor a new address, disregarding when the sample happens. This gives a fair opportunity to short vs. long distance reuses. The probability of an older, already armed watchpoint to be replaced with a new sampled address decays following the harmonic progression [2] over time with each new PMU sample; meaning, the longer a watchpoint stays armed, the higher the probability for it to stay even longer. However, a newly sampled address always has a probability to replace an older sample in such a way that any sampled address has equal probability to be observed irrespective of when the address was sampled. ALLOWREPLACEMENT on Line 19 in Algorithm 1 probabilistically answers this question, whether the sampling thread is allowed to arm watchpoint in a slot in all threads including replacing any old existing untrapped watchpoints. We refer the reader to Reference [58] for theoretical guarantees of reservoir sampling [60, 61].

4.2 SHARED CACHE PROFILING

To detect *reuses* in the shared cache, REUSETRACKER again samples load and store accesses, and employs debug registers to trap accesses to the sampled addresses that happen in the same shared cache.

In this algorithm, when a thread samples a load or a store access, this sampled event is considered as a *use*. After extracting the effective address of the event, the sampling thread arms watchpoints on the address in all threads. Watchpoints that trap load or store accesses are created in cores that share the same socket (and hence the same shared cache) with the sampling thread. In cores that do not share the same socket, watchpoints that trap only store accesses are set up. When the next trap happens in a watchpoint in a core on the same socket, a *reuse* in the shared cache is detected, otherwise an invalidation is detected if the trap happens in a core on another socket.

Fig. 2. One possible execution scenario in profiling reuse distance in L3 cache: (1) Every thread sets its PMUs to sample its load and store accesses. (2) Thread T_1 's PMU counter overflows on a store or a load on address m_1 . (3) T_1 arms the watchpoints on other cores that share the same L3 cache with itself with type WP_RW and with type WP_WRITE on cores that do not share the same L3 cache. (4) T_2 accesses address m_1 again before any other thread, the debug register traps, time reuse distance in L3 is computed. (5) Cache line invalidation in L3 level occurs if T_3 or T_4 stores to address m_1 before T_2 accesses m_1 .

Figure 2 shows the main components and one possible execution scenario of the SHARED CACHE PROFILING algorithm. Next, we explain the steps how REUSETRACKER handles a PMU sample when profiling reuse distance in shared cache.

On a PMU sample: When a thread, say T_1 , samples a load or a store access to an address, say M_1 , a signal is delivered by the Linux kernel to T_1 . This signal is handled by our signal-handling function, which is very similar to the intra-thread profiling algorithm except for few differences. If T_1 is allowed to arm watchpoints globally, then T_1 collects the sum of the PMU counter values of load and store events from all cores that share the same socket as the sampled core. T_1 sets the watchpoint types and arms watchpoints that trap load or store accesses in all of the other threads that share the same socket with itself. The aim of these watchpoints is to detect a reuse in the shared cache. To detect cache line invalidation from the shared cache, T_1 also arms watchpoints that trap only store accesses in cores of other sockets.

On watchpoint trap: When a watchpoint trap happens in a thread, say T_2 , due to an access to an address, say M_1 , the trap is handled by REUSETRACKER. First, T_2 checks if the watchpoint slot is still active to see if no trap has occurred on this slot globally. On an active slot, T_2 checks whether it shares the same socket with the thread that arms its watchpoint. If so, then a shared cache *reuse* is detected and time reuse distance between the trap in T_2 and the sample owner is computed. However, if T_2 and the sample owner are on different sockets, then an invalidation is detected. After detecting a *reuse* or an invalidation, the reservoir sampling probability is reset, and the watchpoint slot is marked as inactive.

Measuring Time Reuse Distance in Shared Cache Profiling: To measure time reuse distance when a *reuse* in a shared cache is detected, we leverage the PMU counter for loads and stores. The time reuse distance is measured as follows.

- (1) At the sampling point when the watchpoints are armed, the PMU sample handler function memorizes the total count $C_1 = \sum_{i=1}^{n_{cores\ on\ skt}} Loads(i) + Stores(i)$ of load and store events from all cores that share the same socket where the sample occurs. C_1 becomes an attribute in each watchpoint that can be accessed by the watchpoint trap handler.
- (2) When a *reuse* is detected, and the trap handler records the total count, say $C_2 = \sum_{i=1}^{n_{cores\ on\ skt}} Loads(i) + Stores(i)$ of loads and stores again from all cores sharing the same socket again. $C_2 - C_1$ is recorded as time reuse distance.

4.3 Implementation

REUSETRACKER is built on top of the open-source HPCToolkit tools suite [4]. It intercepts thread creation and termination using LD_PRELOAD for dynamically linked executables [39]. On each thread creation, we configure the Linux perf_events for the newly created thread to monitor relevant PMU events.

To profile reuse distance in individual threads and in shared caches, we sample MEM_UOPS_RETIRED:ALL_STORES and MEM_UOPS_RETIRED:ALL_LOADS events, respectively, for sampling stores and loads. All of these events allow the signal-handling function to record the accessed effective address and the program counter when an event is sampled. By recording effective address, REUSETRACKER's signal-handling function is able to detect a memory access *use* and its *reuse*.

In addition to recording effective address and calculating time reuse distance, the signal-handling and the watchpoint trap-handling functions also record program counters at the moments of PMU samples and watchpoint traps. Using the program counters, the profiling functions can trace the sampled and trapped threads' call stacks through an online binary analysis. This online binary analysis is performed by HPCToolkit every time a PMU sample or a watchpoint trap is handled to retrieve the procedure frames that are parts of the call stack that becomes the execution context of the detected PMU sample or the watchpoint trap. After tracing the call stack of a sampled or a trapped memory access, the detected *use* or *reuse* can be attributed to the call stack, which makes it easier for programmers to spot the locations of each detected use-reuse pairs in the source code.

Concurrency Control: GlobalWP is a shared data structure and accessed concurrently by different PMU sample handlers and watchpoint traps. We use a two-counter-based transactional memory mechanism proposed by Lamport [31] that allows multiple readers and a single writer to concurrently access the same slot of GlobalWP; accessing different slots of GlobalWP by different threads is obviously conflict free. We ensure mutual exclusion among multiple writers to the same slot via a test-and-test-and-set lock [44].

Workaround for Limited GlobalWP Slot Count: We acknowledge the possibility of having much higher thread count than there are GlobalWP slots. If profiled threads exhibit uniform behavior, then a low number of GlobalWP slots, which is four in x86 architecture, is sufficient. In case there are more than four different behaviors at the same time, REUSETRACKER might still capture all of these behaviors in one run as multiple threads can context switch to be monitored by a GlobalWP slot. However, in an extreme condition where the thread count in an application is much higher than the number of GlobalWP slots and the behaviors of the threads are very diverse to the point that each thread might have its own distinct behavior, there is a possibility that REUSETRACKER might not capture all of these behaviors as it can only monitor four behaviors at the same time. To work around this limitation, a user of our tool might have to profile the application a number of times to get a more complete insight into the reuse behavior of the application.

5 EVALUATION

This section evaluates the accuracy of REUSETRACKER and demonstrates its functionality in analyzing reuse distances in PARSEC, Rodinia, and Synchrobench benchmarks. Additionally, this section also evaluates the performance and memory overheads of REUSETRACKER.

The experimental study is carried out in a two-socket Intel Xeon Gold 6148 Skylake CPU. Each socket has twenty cores, each core has its own L1d, L1i, and L2 caches, and each socket has one shared L3 cache. The machine runs Linux 5.5.2 kernel, and we use gcc 8.3.1 compiler. Unless

```

1 #pragma omp parallel shared(shared_array) \
2   private(private_array1, private_array2,
3   private_array3, \ private_array4, private_array5)
4 {
5   for(int i = 0; i < outer; i++)
6     for(int j = 0; j < a; j++)
7       for(int k = 0; k < inv; k++)
8         Store to shared_array[k]
9       for(int k = 0; k < a1; k = k+2)
10        Load from private_array1[k]
11        Store to private_array1[k+1]
12    for(int j = 0; j < b; j++)
13      for(int k = 0; k < inv; k++)
14        Store to shared_array[k];
15      for(int k = 0; k < b1; k = k+2)
16        Load from private_array2[k]
17        Store to private_array2[k+1]
18    for(int j = 0; j < c; j++)
19      for(int k = 0; k < inv; k++)
20        Store to shared_array[k]
21      for(int k = 0; k < c1; k = k+2)
22        Load from private_array3[k]
23        Store to private_array3[k+1]
24    for(int j = 0; j < d; j++)
25      for(int k = 0; k < inv; k++)
26        Store to shared_array[k];
27      for(int k = 0; k < d1; k = k+2)
28        Load from private_array4[k]
29        Store to private_array4[k+1]
30    for(int j = 0; j < e; j++)
31      for(int k = 0; k < inv; k++)
32        Store to shared_array[k]
33      for(int k = 0; k < e1; k = k+2)
34        Load from private_array5[k]
35        Store to private_array5[k+1]
36 }

```

Listing 1. Pseudo-code for Reuse-Invalidation Benchmark, *RIBench*.

Table 4. Reuse Distance and Reuse Count of *RIBench*

Reuse Distance	Reuse Count
$inv+a1$	$num_threads \cdot outer \cdot a \cdot a1$
$inv+b1$	$num_threads \cdot outer \cdot b \cdot b1$
$inv+c1$	$num_threads \cdot outer \cdot c \cdot c1$
$inv+d1$	$num_threads \cdot outer \cdot d \cdot d1$
$inv+e1$	$num_threads \cdot outer \cdot e \cdot e1$

Table 5. Parameter Values of *RIBench* when Assessing Accuracy without Cache Line Invalidation

Test Case	Parameter Values
Short-RD Increasing	outer = 10, a = 20, a1 = 1,000, b = 20, b1 = 2,000, c = 20, c1 = 4,000, d = 20, d1 = 8,000, e = 20, e1 = 16,000, inv = 0
Short-RD Decreasing	outer = 10, a = 200, a1 = 1,000, b = 60, b1 = 2,000, c = 15, c1 = 4,000, d = 4, d1 = 8,000, e = 2, e1 = 16,000, inv = 0
Short-RD Bell-shaped	outer = 10, a = 50, a1 = 1,000, b = 40, b1 = 2,000, c = 40, c1 = 4,000, d = 10, d1 = 8,000, e = 3, e1 = 16,000, inv = 0
Short-RD Multi-modal	outer = 10, a = 100, a1 = 1,000, b = 30, b1 = 2,000, c = 24, c1 = 4,000, d = 8, d1 = 8,000, e = 7, e1 = 16,000, inv = 0
Long-RD Increasing	outer = 10, a = 10, a1 = 100,000, b = 10, b1 = 200,000, c = 10, c1 = 400,000, d = 10, d1 = 800,000, e = 10, e1 = 1,600,000, inv = 0
Long-RD Decreasing	outer = 10, a = 100, a1 = 100,000, b = 30, b1 = 200,000, c = 12, c1 = 400,000, d = 5, d1 = 800,000, e = 2, e1 = 1,600,000, inv = 0
Long-RD Bell-shaped	outer = 10, a = 50, a1 = 100,000, b = 40, b1 = 200,000, c = 40, c1 = 400,000, d = 10, d1 = 800,000, e = 3, e1 = 1,600,000, inv = 0
Long-RD Multi-modal	outer = 10, a = 100, a1 = 100,000, b = 30, b1 = 200,000, c = 24, c1 = 400,000, d = 8, d1 = 800,000, e = 7, e1 = 1,600,000, inv = 0

otherwise stated, the default sampling interval for both load and store events is 100K, and the used debug register count in each core is four. In each experiment, the threads are distributed evenly across the two sockets, and the threads in each socket are bound to CPU cores with *compact* mapping³ strategy by default.

5.1 *RIBench* Benchmark

To evaluate the accuracy of REUSETRACKER, we develop a synthetic benchmark with controllable reuse and invalidation counts shown in Listing 1. We refer to this benchmark as *RIBench*. Using the *RIBench*, one can configure the amount of reuses for different reuse distances as well as the amount of invalidations that happen during execution.

In the *RIBench*, there are several configurable parameters; *outer*, *a*, *a*₁, *b*, *b*₁, *c*, *c*₁, *d*, *d*₁, *e*, *e*₁, and *inv*. These parameters can be configured to generate expected reuse distances and number of reuses for each reuse distance. There are five different reuse distances that can be generated. These reuse distances are a_1+inv , b_1+inv , c_1+inv , d_1+inv , and e_1+inv . When we want to configure each thread to have five different non-zero reuse distances without being interrupted by any invalidation, we set up the values of *a*₁, *b*₁, *c*₁, *d*₁, and *e*₁ to be five different non-zero values as these five variables determine the number of iterations to access five different

³Compact mapping assigns the thread $t + 1$ to a free thread context as close as possible to the thread context where the thread t was placed.

Fig. 3. Reuse distance histograms of *RIBench* without cache line invalidation. X-axis shows the reuse distance ranges in logarithm-scale. Y-axis displays the fraction of reuse-pairs that belong to specific reuse distance ranges.

private arrays, while the value of *inv* variable is set to be zero. Then, in case we want to introduce cache line invalidations, we set the value of *inv* variable to be a non-zero value, since this variable determines the number of iterations that access a shared array. The other parameters, i.e., *outer*, *a*, *b*, *c*, *d*, and *e*, can be configured to determine the reuse count of each reuse distance.

The computation of expected reuse count for each reuse distance is displayed in Table 4. By changing the parameters, it is possible to produce a variety of reuse distance histogram patterns. For example, if $e_1 > d_1 > c_1 > b_1 > a_1$, $inv = 0$, and $e * e_1 < d * d_1 < c * c_1 < b * b_1 < a * a_1$, the reuse distance histogram will exhibit a decreasing pattern. Another example is when $e_1 > d_1 > c_1 > b_1 > a_1$, $inv = 0$, and $e * e_1 > d * d_1 > c * c_1 > b * b_1 > a * a_1$, then the reuse distance histogram will form an increasing pattern.

In addition to controlling the shapes of patterns in generated reuse distance histograms, it is also possible to control the amount of cache line invalidations by setting up the value of *inv* parameter when there are more than one thread. This parameter determines the iteration count of a loop that performs store access to a shared array *shared_array*. The number of cache line invalidations can be increased by increasing the value of *inv* variable. Therefore, say, for reuse distance $inv+a_1$, instead of having reuse count $num_threads * outer * a * (a_1 + inv)$, its reuse count becomes $num_threads * outer * a * a_1$, as each access to *shared_array* is expected to lead to a cache line invalidation instead of a reuse.

To evaluate the accuracy of REUSETRACKER, we compute the expected reuse distance histogram of the *RIBench* with a given set of parameters, and this histogram is compared with the histogram produced by REUSETRACKER. Given two histograms H and $\hat{H}$, the accuracy, S , is calculated as below [50, 60]:

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

B_i and $\hat{B}_i$ are fractions of reuse-pairs in H and $\hat{H}$, respectively, that fall into the i th bin, and n is the number of bins in each histogram. The value of S metric is in the range $[0, 1.0]$ with 1.0 meaning H and $\hat{H}$ are perfectly similar.

Table 6. Parameter Values of *RIBench* when Assessing Accuracy with Cache Line Invalidations on *bell-shaped* Pattern

Test Case	Parameter Values
Short-RD Bell-shaped Case 1	outer = 10, a = 50, a1 = 1,000, b = 40, b1 = 2,000, c = 40, c1 = 4,000, d = 10, d1 = 8,000, e = 3, e1 = 16,000, inv = 0
Short-RD Bell-shaped Case 2	outer = 10, a = 50, a1 = 500, b = 40, b1 = 1,500, c = 40, c1 = 3,500, d = 10, d1 = 7,500, e = 3, e1 = 15,500, inv = 1,000
Short-RD Bell-shaped Case 3	outer = 10, a = 50, a1 = 0, b = 40, b1 = 1,000, c = 40, c1 = 3,000, d = 10, d1 = 7,000, e = 3, e1 = 15,000, inv = 2,000
Long-RD Bell-shaped Case 1	outer = 10, a = 50, a1 = 100,000, b = 40, b1 = 200,000, c = 40, c1 = 400,000, d = 10, d1 = 800,000, e = 3, e1 = 1,600,000, inv = 0
Long-RD Bell-shaped Case 2	outer = 10, a = 50, a1 = 50,000, b = 40, b1 = 150,000, c = 40, c1 = 350,000, d = 10, d1 = 750,000, e = 3, e1 = 1,550,000, inv = 100,000
Long-RD Bell-shaped Case 3	outer = 10, a = 50, a1 = 0, b = 40, b1 = 100,000, c = 40, c1 = 300,000, d = 10, d1 = 700,000, e = 3, e1 = 1,500,000, inv = 200,000

Table 7. Parameter Values of *RIBench* when Assessing Accuracy with Cache Line Invalidations on *decreasing* Pattern

Test Case	Parameter Values
Short-RD Decreasing Case 1	outer = 10, a = 200, a1 = 1,000, b = 60, b1 = 2,000, c = 15, c1 = 4,000, d = 4, d1 = 8,000, e = 2, e1 = 16,000, inv = 0
Short-RD Decreasing Case 2	outer = 10, a = 200, a1 = 500, b = 60, b1 = 1,500, c = 15, c1 = 3,500, d = 4, d1 = 7,500, e = 2, e1 = 15,500, inv = 1,000
Short-RD Decreasing Case 3	outer = 10, a = 200, a1 = 0, b = 60, b1 = 1,000, c = 15, c1 = 3,000, d = 4, d1 = 7,000, e = 2, e1 = 15,000, inv = 2,000
Long-RD Decreasing Case 1	outer = 10, a = 200, a1 = 100,000, b = 60, b1 = 200,000, c = 15, c1 = 400,000, d = 4, d1 = 800,000, e = 2, e1 = 1,600,000, inv = 0
Long-RD Decreasing Case 2	outer = 10, a = 200, a1 = 50,000, b = 60, b1 = 150,000, c = 15, c1 = 350,000, d = 4, d1 = 750,000, e = 2, e1 = 1,550,000, inv = 100,000
Long-RD Decreasing Case 3	outer = 10, a = 200, a1 = 0, b = 60, b1 = 100,000, c = 15, c1 = 300,000, d = 4, d1 = 700,000, e = 2, e1 = 1,500,000, inv = 200,000

5.2 Accuracy without Invalidation

The simplest case is that of intra-thread reuse distance when there is no invalidation (this is analogous to single-threaded reuse distance). For this base case, we configured the parameters of the *RIBench* to generate four different reuse distance patterns, *increasing*, *decreasing*, *bell-shaped*, and *multi-modal*. We also created two cases for each pattern, short reuse distance (*short-RD*) and long reuse distance (*long-RD*) cases. In the *short-RD* cases, the reuse distance is lower than the sampling period, while in the *long-RD* cases, the reuse distance is higher than or equal to the sampling period. Therefore, in total there are eight different test cases. Table 5 shows the values of the parameters in each test case. As can be seen in the table, *a1*, *b1*, *c1*, *d1*, and *e1* that determine the reuse distances in each thread are always less than the sampling interval, which is 100K, for the *short-RD* cases, while they are always equal to or higher than the sampling interval for the *long-RD* cases.

The results produced by REUSETRACKER running these test cases with 32 threads are presented in Figure 3. As can be seen in the figures, the reuse distance histograms generated by REUSETRACKER are close to the ground truth. The average accuracy for short reuse distance cases is 94%, and the accuracy for long reuse distance cases is 90%. In short reuse distance cases, all reuses can be trapped without having to rely on reservoir sampling. Thus, REUSETRACKER exhibits better accuracy in these cases than in the long reuse distance ones.

Among the long reuse distance cases, we can see that the *Long-RD Decreasing* (Figure 3(f)) and the *Long-RD Multi-modal* (Figure 3(h)) cases have the lowest accuracies. These results can be attributed to the probabilistic nature of reservoir sampling and the fact that the *Long-RD Decreasing* and the *Long-RD Multi-modal* cases have fewer sample counts taken from the loops in lines 18–35 of Listing 1, which access large arrays, than the other two *Long-RD* cases.

5.3 Accuracy with Invalidation

Next, we introduce invalidations in the setup and assess the intra-thread reuse profiling algorithm with cache line invalidations. This setup has 12 test cases that were derived from the *Bell-shaped* and *Decreasing* configurations given in Table 5. Each configuration was modified to generate three different reuse distance histogram patterns that have different amount of cache line invalidations. Table 6 displays the modified configurations for the *Bell-shaped* cases, while Table 7 shows the modified configurations for the *Decreasing* cases. Among these configurations, *Case 3* leads to more cache line invalidations than *Case 2*, which leads to more cache line invalidations than *Case 1*. The

Fig. 4. Reuse distance histograms of *RIBench* with cache line invalidations on *bell-shaped* pattern.

Fig. 5. Reuse distance histograms of *RIBench* with cache line invalidations on *decreasing* pattern.

reason for this is that there are more loop iterations that perform store accesses to `shared_array` in *Case 3* than in *Case 2* and there are more store accesses to the shared array in *Case 2* than in *Case 1*. These store accesses to the shared array is also the main difference between this experiment and the experiment without invalidation in Section 5.2. In the experiment without invalidation, the parameter *inv*, which defines number of store accesses to the shared array, is always zero.

Figures 4 and 5 show the reuse distance histograms generated by REUSETRACKER running on 32 threads compared to the ground truth. As shown in both figures, our results are close to the ground truth with an average accuracy 89% over short reuse distance cases, and 93% over long reuse distance cases. The lower accuracy in short reuse distance cases is the result of our relaxed definition of reuse. Within a short interval between when a thread samples a memory access and when the thread arms watchpoints in all threads, a cache line invalidation to the sampled address

Fig. 6. Accuracy of REUSETRACKER running the *RIBench* under different thread counts. X-axis displays the thread counts, and Y-axis shows the accuracy for each thread count.

might occur undetected. If the interval between the watchpoint arming and the *reuse* of the sampled address is very short, which is the case for the short reuse distance cases, then the undetected invalidation that happens between the sample and the watchpoint arming might be the only opportunity to detect cache line invalidation, which is missed. As a result, a false detection of reuse will occur, which still satisfies our relaxed definition of reuse but violates the ground truth definition in Definition 2.1.

One pattern that can be observed from the ground truths in Figures 4 and 5 is that the offsets of the histograms shift to the right as we go from *Case 1* to 3. The reason for the offset shift is the sum of $inv+al$, which is the shortest reuse distance in all cases, that keeps increasing from *Case 1* to 2 and from *Case 2* to 3. Another pattern that can be observed from the ground truths and the results is that the histograms become narrower as we go from *Case 1* to 3. This is the result of the logarithmic scaling of the bin sizes whose range increases along the X-axis. Therefore, as the histogram shifts to the right due to increases in reuse distances, the capacity of each bin becomes larger, and hence the number of filled bins becomes smaller. The increase in all reuse distances and the logarithmic scaling are also the reasons for the mode shift of the histograms in Figure 4 as the mode that is in the third bin in *Case 1* becomes “pushed” to the fourth bin in *Case 2* and grouped together with the reuses that are already in the fourth bin in *Case 1*. Such transformation also happens from *Case 2* to 3 as some reuses that belong to the fourth bin in *Case 2* are “pushed” to the fifth bin in *Case 3* and makes the height of the fifth bin taller.

5.4 Accuracy under Different Thread Counts

To evaluate REUSETRACKER’s accuracy under different thread counts, we ran it on the *RIBench* with *Long-RD Bell-shaped Case 2* and *Long-RD Decreasing Case 2* configurations. We chose these configurations as they include invalidations and reservoir sampling in their executions and, among all *Long RD* cases that have invalidations, they have the most complex patterns, since they have more filled bins and more variation of tall and short bins in their patterns. We performed the experiment under six different thread counts; 1, 2, 4, 8, 16, and 32. The accuracy results are presented in Figure 6. The accuracy of REUSETRACKER is consistently high under different thread counts with an average of 95% for the *Long-RD Bell-shaped Case 2* and an average of 96% for the *Long-RD Decreasing Case 2*.

5.5 Reuse Distances of PARSEC Benchmarks

In this section, we present and discuss the reuse distance histograms for the PARSEC benchmark suite [9]. For the sake of brevity, we will only discuss four of the benchmarks in detail.

Fig. 7. Histograms of intra-thread reuse distance of *blackscholes* and *bodytrack* from PARSEC. X-axis shows the reuse distance ranges in logarithm-scale. Y-axis displays in logarithm-scale the fraction of reuse-pairs that belong to specific reuse distance ranges.

Figure 7 shows the histograms of *blackscholes* and *bodytrack* generated from intra-thread reuse distance profiling for 32 threads processing *native* input size. Figure 7 shows that most reuses in the selected benchmarks are short in distance. *blackscholes*, which computes prices of a portfolio of European options, has a huge portion of short-distanced reuses. These short-distanced reuses mostly occur on the *prices* array, a data structure that records options' prices. Reuses happen in the *bs_thread*, the *BlkSchlsEqEuroNoDiv*, and the *CNDF* functions. Longer-distanced reuses are detected in the *main* function on accesses to local variables, which become memory accesses due to register spilling.

In *bodytrack*, a lot of reuses of various distances happen within the *TrackingModel::LogLikelihood* function and in other functions called by it. *TrackingModel::LogLikelihood* is a function that computes the likelihood of each observed particle in tracking an object in an image. This computation is needed in resampling particles with the best fitness values from image data to help analyze interesting regions in the image in more detail.

Figure 8 displays the histograms of *streamcluster* and *freqmine* generated from shared cache reuse distance profiling. *streamcluster* is a benchmark that solves an online clustering problem by grouping streamed data points into their nearest centers. In *streamcluster*, some of the detected temporal and spatial reuses happen with distances less than or equal to 8M. Most of these use-reuse pairs happen on *Point*-typed shared array elements in the *pgain*, *dist*, and *shuffle* functions. Aside from accesses to *Point*-typed data structures, there are also detected short-distanced reuses due to locking mechanism in the *pthread_barrier_wait* function. For reuses that are longer than 8M in distance, they also occur because of accesses to shared *Point* data structures in the *dist* function. These reuses can be long in distance, because *dist* is a function that measures distances between data points when making new cluster centers and consecutive accesses to the same data points can happen across different *dist* function calls separated by wide time gaps.

freqmine is a data mining benchmark that performs the **Frequent Pattern-growth (FP-growth)** method for **frequent itemset mining (FIMI)** problem. In this benchmark, reuses of various distances are detected between the *FP_tree::database_tiling* and *FP_tree::scan2_DB* functions due to memory accesses to a shared tree data structure. Furthermore, long-distanced reuses are also detected in the *FP_tree::database_tiling* function on accesses to the *item_order* array. These long-distanced reuses occur due to accesses to the shared array in creating a prefix tree whose branches represent frequent itemsets.

5.6 Use Case: False Sharing Removal

Spatial reuses in shared cache detected by REUSETRACKER include false sharing as the use-reuse pair belong to different threads that access different memory regions located in the same cache

Fig. 8. Histograms of reuse distance in L3 cache of *streamcluster* and *freqmine* from PARSEC. X-axis shows the reuse distance ranges in logarithm-scale. Y-axis displays in linear scale the fraction of reuse-pairs that belong to specific reuse distance ranges.

line. Therefore shared cache spatial reuse distance profiles produced by REUSETRACKER can assist in guiding code refactoring that reduces false sharing in the profiled code. To demonstrate this capability, we leverage REUSETRACKER to profile some benchmarks from Synchrobench [22], namely, *ESTM-rbtree*, *MUTEX-hashtable*, *MUTEX-lazy-list*, *SPIN-hashtable*, and *SPIN-lazy-list*, and use the generated profiles to guide code modifications that improve their performances. To detect only false sharing, we generate a shared cache reuse distance profile that includes only **read-after-write (RAW)** and **write-after-write (WAW)** use-reuse pairs. The reason for this is these kinds of use-reuse pairs trigger inter-core cache line transfers caused by false sharing for certain.

In *ESTM-rbtree*, we identify false sharing in the `TMlookup` function that involves `struct node` and `struct thread_data` data types. To remove the false sharing, we inserted padding into `struct node` and `struct thread_data`. After this modification, *ESTM-rbtree*'s performance is improved by 87%. In *MUTEX-hashtable*, *MUTEX-lazy-list*, *SPIN-hashtable*, and *SPIN-lazy-list*, false sharing is found in the `parse_delete` function or in other functions called from it. In all of these benchmarks, the detected false sharing involves `struct node_1` data type. Paddings in this data structure improves the performance of these benchmarks by 6%, 46%, 39%, and 58% for *MUTEX-hashtable*, *MUTEX-lazy-list*, *SPIN-hashtable*, and *SPIN-lazy-list*, respectively.

5.7 Use Case: Adjacent Cache Line Prefetch

By analyzing the intra-thread and shared cache reuse distance histograms of a multi-threaded code, we demonstrate that it is possible to decide whether the code will benefit from ACP [23] or not. ACP, a feature in Intel microarchitectures, allows prefetching a cache line that is adjacent to the currently accessed cache line. This feature can improve an application performance if the application threads have a good spatial locality where each thread actually accesses the prefetched cache lines before they are evicted. In addition to locality in individual threads, another factor that affects application performance when using ACP is inter-thread communication. Excessive communication, such as false sharing, may hinder the benefit of spatial locality that ACP offers. Due to communication, prefetched cache lines could be invalidated before they are accessed by the prefetching threads. To account for locality-affected access latencies and inter-thread communication, we build a model to predict whether an application can benefit from ACP or not.

We consider an application to have better performance with ACP if its performance overhead due to main memory access is higher than its overhead due to false sharing. Let G be a binary predictor metric, where

$$G = T_m - T_{fs},$$

such that T_m is the total latency of accesses to main memory, and T_{fs} is the total false sharing latency. T_m in this model is computed as follows:

$$T_m = R_{dram} * L_{dram},$$

where R_{dram} is the number of detected spatial reuses in the intra-thread reuse distance histogram that access DRAM, and L_{dram} is the latency of access to DRAM. In the model, T_{fs} is calculated as follows:

$$T_{fs} = R_{fs} * L_{fs},$$

where R_{fs} is the number of detected spatial reuses in the shared cache reuse distance histogram with reuse distances that are short enough to be in the range of false sharing communication, and L_{fs} is the latency of inter-core cache line transfer. We obtained the values of L_{dram} and L_{fs} by running **Intel Memory Latency Checker (Intel MLC)** [57] on our machine.

Table 8. Prediction of Execution Outcomes When ACP Is Activated

Benchmarks	Predictor Metric	Actual Speedup
bodytrack	$-1.1 * 10^9$ ($G < 0$)	-5.41%
streamcluster	$-1.2 * 10^8$ ($G < 0$)	-8.84%
swaptions	0 ($G \geq 0$)	9.74%
vips	0 ($G \geq 0$)	12.02%
backprop	$6.9 * 10^{10}$ ($G \geq 0$)	8.21%
bfs	$7.6 * 10^{10}$ ($G \geq 0$)	15.54%
ESTM-specfriendly-tree	$-1.6 * 10^{10}$ ($G < 0$)	-25.00%
lockfree-fraser-skiplist	$-1.5 * 10^{12}$ ($G < 0$)	-12.10%
MUTEX-hashtable	$-3.1 * 10^{11}$ ($G < 0$)	-7.25%
MUTEX-skiplist	$-9.6 * 10^{10}$ ($G < 0$)	-15.34%
SPIN-hashtable	$-5.3 * 10^{10}$ ($G < 0$)	-11.66%
SPIN-hoh-list	$-1.8 * 10^{10}$ ($G < 0$)	-11.24%

We expect an application to get a performance speedup when running with ACP if $G \geq 0$. By using this metric, we predict whether an application will gain or lose performance when ACP is activated in the Intel Xeon Gold 6148 Skylake. The benchmarks that we use in this experiment are 10 benchmarks from PARSEC [9] (*bodytrack*, *streamcluster*, *swaptions*, *vips*, *blackscholes*, *dedup*, *facesim*, *ferret*, *fluidanimate*, and *fraqmine*), 7 benchmarks from Rodinia [12] (*backprop*, *bfs*, *hotspot*, *kmeans*, *leukocyte*, *needle*, and *srad*), and all 14 benchmarks from Synchrobench [22]. After running these benchmarks with ACP enabled, we noticed that only 12 of them display performance gains or losses that are higher than 5%. Our model accurately predicts the execution outcomes of these 12 benchmarks as shown in Table 8.

5.8 Overhead Analysis

To evaluate the runtime and memory overheads of REUSETRACKER, we run it on ten PARSEC benchmarks to profile their intra-thread reuse distance with 100K sampling interval. For these benchmarks, the average runtime overhead is 2.9×, and the average memory overhead is 2.8×, which are much lower than the overheads of existing simulator and binary instrumentation-based tools as shown in Table 1. However, the overheads of REUSETRACKER are slightly higher than the overheads of other PMU-based techniques like StatCache [7] and RDX [60]. This is because each sampling thread in REUSETRACKER arms not only its own debug registers but also debug registers of other cores. The comparison between REUSETRACKER and other similar techniques are shown in Table 1. The overheads of RDX and loca were obtained by running them with the same ten PARSEC benchmarks, and RDX with 100K sampling interval. The overheads of RDX reported here are higher than the ones reported in its paper, because in Reference [60] authors used a sampling interval of 5M. In addition to measuring overheads while profiling reuse distance in individual threads, we also measure overheads for shared cache profiling. The average runtime overhead for reuse distance profiling in shared cache is 2.1×, and the average memory overhead is 2.4×, which are even lower than the overheads of intra-thread profiling. These lower overheads might be caused by lower number of watchpoint traps that are handled at shared cache level than at intra-thread level.

6 RELATED WORK

Existing related works can be organized into two groups. The first group consists of the papers that proposed multi-core reuse distance analysis techniques that profile individual threads and shared caches. The second group introduced reuse distance analysis techniques that leverage PMUs

and debug registers. Table 1 presents the comparison between REUSETRACKER and some of the techniques discussed in this section that also perform detection of *reuses* at runtime.

6.1 Modeling Individual Threads and Shared Caches

Ding and Chilimbi proposed an analytical model that predicts shared cache behavior indirectly by combining per-thread reuse distance, interleaving, and data sharing information [14]. This technique processes execution trace of a multithreaded code to generate a set of per-thread metrics. This set of metrics cover locality (i.e., reuse distance profile), data sharing, and interleaving information of each thread. By accounting for multiple sets of metrics from a subset of threads using the model, miss ratio of this subset of threads while running in a shared cache can be predicted. Another work by Jiang et al. [29] proposed a probabilistic model that can approximate concurrent reuse distance from reuse distances of individual threads. The multithreaded applications targeted by this model are those that perform similar computations in all threads and have uniform amount of shared data across their thread groups. These techniques in References [14, 29] require processing of execution traces that might incur large storage overhead in the host machines. Furthermore, the computation of reuse distance profiles from the traces can also require huge performance overhead depending on the size of the traces.

Schuff et al. introduced a different approach in Reference [48]. Their technique performs on-the-fly reuse distance computations by using a modified hardware simulator. This technique computes reuse distances in both private and shared caches by also ensuring the coherence in the private caches. Furthermore, it also models the interleaving of memory accesses by all threads in the shared cache. This technique was improved in Reference [47] by having all threads run in parallel while being profiled. Moreover, sampling technique was also introduced to reduce overhead by computing reuse distances only on randomly selected memory references. A shortcoming in these approaches is that they require simulator or binary instrumentation to intercept every single memory access. The use of simulator introduces huge performance and memory consumption overhead. Furthermore, the use of binary instrumentation might distort the parallel schedules of task-parallel and dataflow applications as reported in Reference [41].

Chandra et al. [13] introduced one of the first models that predict L2 cache misses when multiple threads run on a shared L2 cache. Eklov et al. [20] also proposed another model that estimates miss ratio and CPI of co-scheduled applications that run on a shared cache. These models work by firstly approximating the reuse distance profile of the applications' interleaved accesses in the shared cache. After that, the models calculate the shared cache miss ratio based on the approximated profile. One key difference of these works from ours is that they target co-scheduled applications that do not share data among them.

Pericas et al. [41] introduced a low-overhead method to generate execution traces of multithreaded code and compute reuse distances of shared caches from these traces. In generating shared cache reuse distance histograms, it also captures cache coherence-based invalidations across caches. To minimize overhead, this method reduces trace sizes by operating at the granularity of compute kernels. By operating at coarse-grained level, this method can accurately capture distant reuses while losing information on near reuses.

Maeda et al. [36] introduced a technique to profile reuse distances in every level of a multi-level cache hierarchy. Since this technique still requires memory address trace as an input, it still needs other tools to generate the trace. As a result, this technique is still exposed to the drawbacks of these tools, such as huge overhead in simulators or distortion of parallel schedule due to binary instrumentation.

Another work that leverages memory traces to profile reuse distances in private and shared caches was proposed by Barai et al. [5]. In this work, they utilized a compiler-assisted technique

to generate a basic block-labeled memory trace from a single sequential execution of a profiled application. This trace is then used by a probabilistic analytical method to predict the reuse distance profiles of the application in private and shared caches when the application runs in parallel.

Hu et al. [24] proposed a model that reduces time and space costs in constructing cache **miss ratio curves (MRCs)** by analyzing only reuse-time distribution. They used **average eviction time (AET)** as a parameter to detect reuses that lead to cache misses. Their model could be extended to predict cache misses in a shared cache when multiple threads run on it. However, in estimating cache misses in private and shared caches, their model does not assume data sharing among threads that might lead to coherence misses in private caches and shared caches.

Ji et al. [26] developed a probability model that computes L2 reuse distance profile and predicts cache misses in L2 without requiring extra simulations or trace generations. Though able to give cache miss prediction, this model still needs inputs in the form of L1 reuse distance histograms. These inputs might have to be generated by simulator or binary instrumentation-based tools.

There are also recent works in References [33, 34, 46, 59] that proposed analytical models to profile shared cache behaviors by processing L1 cache reuse distance profiles. Similar to previously discussed works, these works also depend on other tools to generate per-thread reuse distance histograms that they need.

6.2 Leveraging PMUs and Debug Registers

To the best of our knowledge, there have been only two techniques in literature that utilize hardware counters and watchpoint mechanism to compute reuse distances. The first technique was presented by Berg and Hagersten in Reference [7]. In that paper, they proposed StatCache, a profiling tool that calculates time reuse distances by using hardware counters and watchpoints and deploys a statistical model to predict cache miss ratio based on the collected time reuse distance profile. This tool could predict the miss ratio of a fully associative cache with random replacement policy. The second technique that leverages hardware counters and watchpoints was introduced by Wang et al. [60]. This technique, which is named RDX, utilizes PMUs and debug registers in Intel machines to compute time reuse distances, and then, convert them to stack reuse distances. Both techniques in References [7, 60] differ from our work in the way that they do not account for inter-thread interactions in multithreaded code and they do not model shared caches.

7 CONCLUSION

In this work, we developed REUSETRACKER, a reuse distance analysis tool that profiles reuse distance in individual threads and in shared caches of multi-threaded applications with low overheads by leveraging PMUs and debug registers. We proposed two different algorithms to profile reuse distance in individual threads and shared caches, respectively. To verify the accuracy of the intra-thread profiling algorithm, we developed a synthetic benchmark that can be configured to generate a variety of reuse distance histogram patterns. We also demonstrated how REUSETRACKER can be used to guide performance optimization by removing false sharing via code refactoring and how it can predict whether multi-threaded applications gain or lose performance when ACP is enabled. By using REUSETRACKER, programmers will be able to profile data locality in thread and shared caches with low overheads, and use the generated information to tune application performance.

REFERENCES

- [1] GitHub. [n.d.]. dcompiler/loca: Program Locality Analysis Tools. Retrieved on 20 July, 2020 from <https://github.com/dcompiler/loca>.
- [2] Wikipedia. [n.d.]. Harmonic Progression. Retrieved on 12 January, 2021 from [https://en.wikipedia.org/wiki/Harmonic_progression_\(mathematics\)](https://en.wikipedia.org/wiki/Harmonic_progression_(mathematics)).

- [3] Intel. [n.d.]. Thread Affinity Interface (Linux* and Windows*). Retrieved on 1 February, 2021 from <https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows>.
- [4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurr. Comput.: Pract. Exper.* 22, 6 (2010), 685–701.
- [5] Atanu Barai, Gopinath Chennupati, Nandakishore Santhi, Abdel-Hameed Badawy, Yehia Arafa, and Stephan Eidenbenz. 2020. PPT-SASMM: Scalable analytical shared memory model: Predicting the performance of multicore caches from a single-threaded execution trace. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'20)*. Association for Computing Machinery, New York, NY, 341–351. <https://doi.org/10.1145/3422575.3422806>
- [6] E. Berg and E. Hagersten. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. 20–27.
- [7] Erik Berg and Erik Hagersten. 2005. Fast data-locality profiling of native execution. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. Association for Computing Machinery, New York, NY, 169–180. <https://doi.org/10.1145/1064212.1064232>
- [8] Kristof Beyls and Erik D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems (IASTED'01)*. 617–622.
- [9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81.
- [10] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*. Association for Computing Machinery, New York, NY, 150–159. <https://doi.org/10.1145/782814.782836>
- [11] Germán Ceballos, Erik Hagersten, and David Black-Schaffer. 2016. Formalizing data locality in task parallel applications. In *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, Cham, 43–61.
- [12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, 44–54. <https://doi.org/10.1109/IISWC.2009.5306797>
- [13] Dhruva Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. 340–351. <https://doi.org/10.1109/HPCA.2005.27>
- [14] Chen Ding and Trishul Chilimbi. 2009. *A Composable Model for Analyzing Locality of Multi-threaded Programs*. Technical Report MSR-TR-2009-107. Retrieved from <https://www.microsoft.com/en-us/research/publication/a-composable-model-for-analyzing-locality-of-multi-threaded-programs/>.
- [15] Chen Ding and Yutao Zhong. 2001. *Reuse Distance Analysis*. Technical Report.
- [16] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.* 38, 5 (May 2003), 245–257. <https://doi.org/10.1145/780822.781159>
- [17] Paul J. Drongowski. 2007. Instruction-based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Retrieved from <https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf>.
- [18] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, 389–400. <https://doi.org/10.1109/MICRO.2012.43>
- [19] David Eklov, David Black-Schaffer, and Erik Hagersten. 2010. StatCC: A Statistical Cache Contention Model. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (Vienna, Austria) (PACT'10)*. Association for Computing Machinery, New York, NY, USA, 551–552. <https://doi.org/10.1145/1854273.1854347>
- [20] David Eklov, David Black-Schaffer, and Erik Hagersten. 2011. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*. Association for Computing Machinery, New York, NY, 147–157. <https://doi.org/10.1145/1944862.1944885>
- [21] D. Eklov and E. Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 55–65.
- [22] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. Association for Computing Machinery, New York, NY, 1–10. <https://doi.org/10.1145/2688500.2688501>

- [23] Ravi Hegde. 2015. Optimizing Application Performance on Intel Core Microarchitecture Using Hardware-Implemented Prefetchers. Retrieved from <https://software.intel.com/content/www/us/en/develop/articles/optimizing-application-performance-on-intel-core-t-microarchitecture-using-hardware-implemented-prefetchers.html>.
- [24] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast miss ratio curve modeling for storage cache. *ACM Trans. Storage* 14, 2, Article 12 (Apr. 2018), 34 pages. <https://doi.org/10.1145/3185751>
- [25] Intel. 2010. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. Retrieved from <https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf>.
- [26] Kecheng Ji, Ming Ling, and Li Liu. 2018/02. A probability model of calculating L2 cache misses. In *Proceedings of the International Conference on Computer Science, Electronics and Communication Engineering (CSECE'18)*. Atlantis Press, 329–332. <https://doi.org/10.2991/csece-18.2018.71>
- [27] Kecheng Ji, Ming Ling, and Longxing Shi. 2017. Using the first-level cache stack distance histograms to predict multi-level LRU cache misses. *Microprocess. Microsyst.* 55, C (Nov. 2017), 55–69. <https://doi.org/10.1016/j.micpro.2017.10.001>
- [28] Kecheng Ji, Ming Ling, Yang Zhang, and Longxing Shi. 2017. An artificial neural network model of LRU-cache misses on out-of-order embedded processors. *Microprocess. Microsyst.* 50 (2017), 66–79. <https://doi.org/10.1016/j.micpro.2017.02.005>
- [29] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*, Rajiv Gupta (Ed.). Springer, Berlin, 264–282.
- [30] Mark Scott Johnson. 1982. Some requirements for architectural support of software debugging. In *Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS'82)*. ACM, New York, NY, 140–148. <https://doi.org/10.1145/800050.801837>
- [31] Leslie Lamport. 1977. Concurrent reading and writing. *Commun. ACM* 20, 11 (Nov. 1977), 806–811. <https://doi.org/10.1145/359863.359878>
- [32] David Levinthal. 2009. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. Retrieved from <https://www.amd.com/system/files/TechDocs/24594.pdf>.
- [33] Ming Ling, Jiancong Ge, and Guangmin Wang. 2020. Fast modeling L2 cache reuse distance histograms using combined locality information from software traces. *J. Syst. Architect.* 108 (2020), 101745. <https://doi.org/10.1016/j.sysarc.2020.101745>
- [34] Ming Ling, Xiaoqian Lu, Guangmin Wang, and Jiancong Ge. 2021. Analytical modeling the multi-core shared cache behavior with considerations of data-sharing and coherence. *IEEE Access* 9 (2021), 17728–17743. <https://doi.org/10.1109/ACCESS.2021.3053350>
- [35] Linux. 2012. perf_event_open—Linux Man Page. Retrieved from https://linux.die.net/man/2/perf_event_open.
- [36] R. K. V. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian. 2017. Fast and accurate exploration of multi-level caches using hierarchical reuse distance. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 145–156.
- [37] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.
- [38] R. E. McLearn, D. M. Scheibelhut, and E. Tammaru. 1982. Guidelines for creating a debuggable processor. In *Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS'82)*. ACM, New York, NY, 100–106. <https://doi.org/10.1145/800050.801833>
- [39] Greg Nakhimovsky. 2001. Debugging and Performance Tuning with Library Interposers. Retrieved from http://dsc.sun.com/solaris/articles/lib_interposers.html.
- [40] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. Association for Computing Machinery, New York, NY, 2–11. <https://doi.org/10.1145/1772954.1772958>
- [41] Miquel Pericas, Kenjiro Taura, and Satoshi Matsuoka. 2014. Scalable analysis of multicore data reuse and sharing. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. Association for Computing Machinery, New York, NY, 353–362. <https://doi.org/10.1145/2597652.2597674>
- [42] V. Reddi, A. Settle, Daniel A. Connors, and R. Cohn. 2004. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education (WCAE'04)*.
- [43] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture (WCAE'04)*. Association for Computing Machinery, New York, NY, 22–es. <https://doi.org/10.1145/1275571.1275600>

- [44] Larry Rudolph and Zary Segall. 1984. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA'84)*. Association for Computing Machinery, New York, NY, 340–347. <https://doi.org/10.1145/800015.808203>
- [45] J. M. Sabarimuthu and T. G. Venkatesh. 2018. Analytical miss rate calculation of L2 cache from the RD profile of L1 cache. *IEEE Trans. Comput.* 67, 1 (2018), 9–15.
- [46] J. M. Sabarimuthu and T. G. Venkatesh. 2019. Analytical derivation of concurrent reuse distance profile for multi-threaded application running on chip multi-processor. *IEEE Trans. Parallel Distrib. Syst.* 30, 8 (2019), 1704–1721.
- [47] D. L. Schuff, M. Kulkarni, and V. S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 53–63.
- [48] D. L. Schuff, B. S. Parsons, and V. S. Pai. 2010. Multicore-aware reuse distance analysis. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. 1–8.
- [49] Rathijit Sen and David A. Wood. 2013. Reuse-based online models for caches. *SIGMETRICS Perform. Eval. Rev.* 41, 1 (June 2013), 279–292. <https://doi.org/10.1145/2494232.2465756>
- [50] Xipeng Shen, Jonathan Shaw, and Brian Meeker. 2006. *Accurate Approximation of Locality from Time Distance Histograms*. Technical Report.
- [51] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality approximation using time. *SIGPLAN Not.* 42, 1 (Jan. 2007), 55–61. <https://doi.org/10.1145/1190215.1190227>
- [52] 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. 2011. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD* 55, 3 (May-June 2011), 4:1–4:19.
- [53] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. Association for Computing Machinery, New York, NY, 392–403. <https://doi.org/10.1145/223982.224449>
- [54] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericas. 2017. Trends in data locality abstractions for HPC systems. *IEEE Trans. Parallel Distrib. Syst.* 28, 10 (Oct. 2017), 3007–3020. <https://doi.org/10.1109/TPDS.2017.2703149>
- [55] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. 2016. TiDA: High-level programming abstractions for data locality management. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 116–135.
- [56] S. Van den Steen and L. Eeckhout. 2018. Modeling superscalar processor memory-level parallelism. *IEEE Comput. Architect. Lett.* 17, 1 (2018), 9–12.
- [57] Krishnaswamy Viswanathan. 2013. Intel Memory Latency Checker v3.9. Retrieved from <https://software.intel.com/content/www/us/en/develop/articles/intel-memory-latency-checker.html>.
- [58] Jeffrey Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw.* 11 (Mar. 1985), 37–57. <https://doi.org/10.1145/3147.3165>
- [59] G. Wang, J. Ge, Y. Yan, and M. Ling. 2019. A data-sharing aware and scalable cache miss rates model for multi-core processors with multi-level cache hierarchies. In *Proceedings of the IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. 267–274.
- [60] Q. Wang, X. Liu, and M. Chabbi. 2019. Featherlight reuse-distance measurement. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE Computer Society, Los Alamitos, CA, 440–453. <https://doi.org/10.1109/HPCA.2019.00056>
- [61] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for software inefficiencies with witch. *SIGPLAN Not.* 53, 2 (Mar. 2018), 332–347. <https://doi.org/10.1145/3296957.3177159>
- [62] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for software inefficiencies with witch. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. Association for Computing Machinery, New York, NY, 332–347. <https://doi.org/10.1145/3173162.3177159>
- [63] M. Wu and D. Yeung. 2011. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 264–275.
- [64] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A higher order theory of locality. *SIGARCH Comput. Archit. News* 41, 1 (Mar. 2013), 343–356. <https://doi.org/10.1145/2490301.2451153>
- [65] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.* 31, 6, Article 20 (Aug. 2009), 39 pages. <https://doi.org/10.1145/1552309.1552310>

Received May 2021; revised June 2021; accepted August 2021