DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling

Tanvir Ahmed Khan
University of Michigan

Ian Neal
University of Michigan

Gilles Pokam
Intel Corporation

Barzan Mozafari
University of Michigan

Baris Kasikci
University of Michigan

Abstract

Poor data locality hurts an application's performance. While compiler-based techniques have been proposed to improve data locality, they depend on heuristics, which can sometimes hurt performance. Therefore, developers typically find data locality issues via dynamic profiling and repair them manually. Alas, existing profiling techniques incur high overhead when used to identify data locality problems and cannot be deployed in production, where programs may exhibit previously-unseen performance problems.

We present selective profiling, a technique that locates data locality problems with low-enough overhead that is suitable for production use. To achieve low overhead, selective profiling gathers runtime execution information selectively and incrementally. Using selective profiling, we build DMon, a system that can automatically locate data locality problems in production, identify access patterns that hurt locality, and repair such patterns using targeted optimizations.

Thanks to selective profiling, DMon's profiling overhead is 1.36% on average, making it feasible for production use. DMon's targeted optimizations provide 16.83% speedup on average (up to 53.14%), compared to a baseline that uses the highest level of compiler optimization. DMon speeds up PostgreSQL, one of the most popular database systems, by 6.64% on average (up to 17.48%).

1 Introduction

Poor data locality is the root cause of many performance problems [6, 34, 48]. Rapidly increasing data footprints of modern applications due to heavily data-driven use cases (*e.g.*, analytics [109], machine learning [1], etc.) make matters worse, precipitating data locality problems further [6]. Recent work shows that up to 64% of all CPU cycles are lost due to poor data locality for widely used data center applications [90].

Although many compiler optimizations aim to eliminate data locality problems statically [3, 22, 23, 70, 71], such optimizations rely on compile-time heuristics, which may not accurately identify and repair problems that manifest dynamically at run time. In fact, as we (§6.2) and others [2, 15, 20, 27] demonstrate, compiler-based techniques can sometimes even hurt performance when the assumptions made by those heuristics do not hold in practice.

To overcome the limitations of static optimizations, the systems community has invested substantial effort in developing dynamic profiling tools [28, 38, 57, 97, 102]. Dynamic profilers are capable of gathering detailed and more accurate execution information, which a developer can use to identify and resolve data locality problems.

Traditionally, existing dynamic profiling tools have been used offline, namely during testing and development, where test cases are designed to adequately represent real-world program behavior. However, due to the proliferation of cloud computing and mobile devices, programs exhibit vast variations in terms of how they execute and consume data in production [48, 84]. Consequently, it has become increasingly difficult for offline profiling to be representative of how programs behave in production settings.

Unfortunately, existing dynamic profilers incur considerable overheads when used to detect data locality issues, and therefore they are not suitable for production environments [13, 57, 60–62, 77, 78].

In this paper, we present *selective profiling*, a data locality profiling technique that not only accurately detects data locality problems, but also incurs low overhead, making it suitable for production deployment. Using selective profiling, we design DMon, a system that can automatically detect and eliminate data locality problems in production systems.

Selective profiling is a lightweight technique to continuously monitor production executions for symptoms of poor data locality (*e.g.*, frequent memory stalls, increased cache misses, etc.). As these high-level indicators of data locality problems are identified, selective profiling automatically transitions to incrementally monitoring more precise information about the source location and exact cause of the data locality problem—this is done by traversing a hierarchical abstraction we introduce, called the *data locality tree* (§3), which allows

DMon to monitor hardware events in a selective way to create an accurate profile at low run-time overhead.

After gathering the profile, DMon performs an offline analysis to identify common patterns of memory accesses. DMon then matches these patterns to a set of existing data locality optimizations (§4.1), which it primarily applies automatically, in a targeted manner (unlike static techniques). For cases where DMon cannot automatically apply an optimization, it provides detailed information about the locality problem to the developer, who can fix the problem manually; in our evaluation, this case occurs only once and the developer can apply DMon-suggested optimization with minimal effort (<10 LOC). We provide four optimization passes (§4.2) which DMon can use to automatically fix data locality problems and are sufficient for DMon to fix major data locality problems we identify across the systems we test in our evaluation (§6).

Selective profiling incurs 1.36% monitoring overhead on average, making it an ideal profiling technique for detecting data locality issues in production. The run-time overhead of selective profiling is significantly (*i.e.*, 9×) lower than that of the state-of-the-art data locality profiler [17, 68]. Overall, targeted optimizations performed by DMon for 13 applications deliver on average 16.83% (up to 53.14%) speedup. To show the effectiveness of DMon for large real-world systems, we applied DMon to `PostgreSQL` [92], a popular open-source database system, where DMon-guided optimizations provided on average 6.64% and up to 17.48% speedup across all 22 TPC-H [26] queries. Furthermore, the optimizations enabled by DMon provides 20% more speedup, on average, than optimizations provided by the same state-of-the-art profiler.

Overall, we make the following contributions:

- Selective profiling, a data locality profiling technique that automatically and incrementally monitors fine-grained execution information to accurately detect data locality problems with low overhead.
- DMon, a system that implements selective profiling to detect data locality problems in production systems. DMon automatically selects specific optimizations based on memory access patterns, and applies these well-known optimization techniques automatically in most cases.
- By evaluating DMon in the context of widely-used applications, we show that selective profiling can detect data locality issues in production with low overhead (1.36% on average). Moreover, we show that selective profile-guided targeted data locality optimizations provide significant performance speedup (16.83% on average, up to 53.14%).

We explain the key design challenge for accurately and efficiently detecting data locality problems in §2. We describe selective profiling in §3, DMon's design in §4, and DMon's implementation in §5. We evaluate DMon in §6, compare DMon to related work in §7, and conclude in §8.

2 Challenges

It is challenging to accurately pinpoint data locality problems, while incurring low run-time performance overhead.

Compiler-based static data locality optimizations [14, 70, 71, 82, 91] are appealing because they incur no run-time overhead. However, static techniques apply optimizations based on compile-time heuristics, which may not accurately identify program locations that suffer from poor locality at run time. In fact, compiler-based techniques can sometimes even hurt performance when the assumptions made by those heuristics do not hold in practice [2, 15, 20, 27].

To demonstrate how compile-time heuristics can hurt performance, we use a compiler-based data prefetching technique [71] to improve data locality in two matrix decomposition benchmarks [104], `lu_cb` and `lu_ncb` from the PARSEC suite [12]. This optimization combines loop splitting and explicit data prefetching to increase data locality. Using the benchmarks' standard inputs, we determine that 50% of all the cache misses in `lu_cb` and `lu_ncb` stem from a single function, which we optimized using compiler-guided data prefetching [71]. The optimization provides a 19.4% speedup for `lu_ncb`, but yields a 19.85% slowdown for `lu_cb`. This occurs because, for `lu_ncb`, prefetching reduces all cache misses; however, for `lu_cb`, there was a dramatic increase in L2 cache misses despite a reduction in L1 and L3 cache misses.

Dynamic profilers can accurately pinpoint data locality problems [13, 57, 60–62, 77, 78], however, they impose considerable overhead (*i.e.*, >10% on average), as they track too much information: memory accesses, timestamps, cache events, etc. Consequently, existing data locality profilers are not deployed in production.

A potential remedy to the high overhead of existing profilers is statistical sampling, which can collect information with reasonable overhead [9]. For instance, the state-of-the-art Intel VTune profiler [85] samples information such as hardware and software performance counters, timestamps, program locations, and accessed memory addresses to gather the necessary information for detecting data locality issues.

Alas, even sampling is not enough to reduce the overhead incurred by popularly available profilers (*e.g.*, Intel VTune) to detect data locality problems to levels acceptable for production use. To assess the impact of sampling, we use the state-of-the-art profiler VTune to detect the data locality issues in our evaluation targets. Despite sampling-based data collection, VTune still incurs 26% overhead on average (and up to 60%), which is unacceptable for production settings.

We argue that not only the monitored execution information must be deliberately chosen to only pertain to data locality problems, but monitoring must occur incrementally, only when there are increasingly clear signs of poor data locality. Next, we explain how selective profiling achieves this.

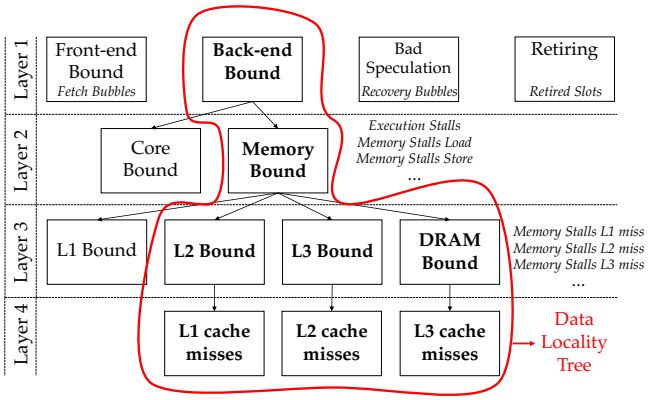

Figure 1: The locality tree abstraction. Performance events that pertain to each tree node are in *italic*. There are no dedicated events to determine if a program is back-end bound. Instead, selective profiling subtracts from total stalls the sum of the stalls that cause other bottlenecks at layer 1 to determine if an execution is back-end bound.

3 Selective Profiling

Selective profiling is a monitoring technique that incrementally monitors more detailed, yet more targeted, run-time information to identify data locality problems. Next, we discuss the three key components of selective profiling: (1) Targeted Monitoring, (2) Incremental Monitoring, and (3) Sampling.

3.1 Targeted Monitoring

Unlike existing offline profilers [57, 97, 98, 102, 106] that monitor many hardware events and information such as program locations, selective profiling needs to carefully choose which information to monitor in order to accurately and efficiently detect data locality problems. A straw-man approach is to only monitor events such as data cache misses, which are directly related to data locality problems. However, simply monitoring data cache misses in isolation can be misleading. For instance, a seemingly large number of data cache misses may have no impact on the performance of an application that spends a lot of time fetching instructions to execute (a common theme in modern Web services [8, 48]).

Selective profiling monitors a select group of hardware events that allow it to determine if the execution of a program is bounded by a subset of those events that we call the *data locality tree*. As shown in Fig. 1, the data locality tree is a hierarchical abstraction of data locality-related performance events from Intel's Top-Down methodology [106]. The Top-Down methodology provides a breakdown of performance events in Intel CPUs, which a developer can use as a guideline to navigate their manual profiling efforts. However, unlike Top-Down, selective profiling automatically transitions from one layer to another, incrementally monitoring more events at each layer of the tree, as increasing evidence of data locality issues is observed at run time.

At layer 1, selective profiling determines whether the execution is back-end bound—*i.e.*, spends a large portion of the time either in CPU execution (CPU bound) or accessing memory (memory bound). At layer 1, a program can also be front-end bound (*i.e.*, fetching instructions), incurring misspeculations, or retiring instructions. For executions that are back-end bound, selective profiling determines whether they are processor-core bound or memory bound in layer 2.

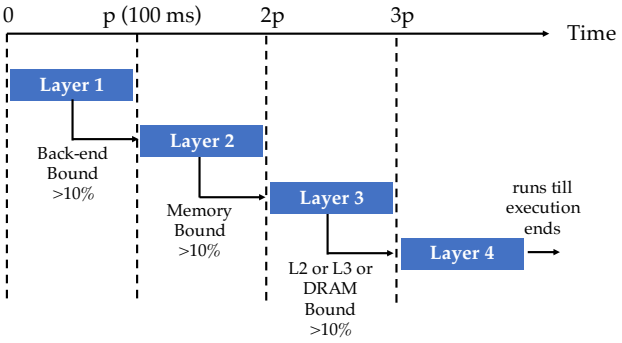

Figure 2: Incremental monitoring

If an execution is memory bound in layer 2, selective profiling monitors events that provide a breakdown of the execution into 4 categories in layer 3. Of those 4 categories, only 3 are related to data locality problems: L2 bound and L3 bound represent the time spent accessing the L2 and the L3 cache, respectively; "DRAM bound" represents the time spent accessing the DRAM. If a program is L1 bound, the data or instructions that the program uses are already as close to the processor as possible and it is hard to improve data locality further. In such cases, the program may have other performance problems, such as false sharing [93] or lock contention [87].

Selective profiling also tracks information to map performance problems back to code. In layer 4, selective profiling records program location information along with hardware events. For example, if a program is L2 bound, selective profiling records L1 cache misses and the location of the instruction causing the miss. By locating and reducing L1 cache misses, the execution time will potentially not be L2 bound, and the locality problem will likely be fixed. Similarly, if a program is L3 or DRAM bound, selective profiling records L2 and L3 cache misses and associated program locations, respectively.

3.2 Incremental Monitoring

Unfortunately, merely restricting the scope of monitored performance events to the data locality tree is not sufficient for low overhead monitoring of data locality issues. Thus, selective profiling instead adopts an incremental monitoring approach. This approach increases the amount of information gathered at run time to efficiently identify program locations that may have a locality problem.

Fig. 2 shows the details of incremental monitoring. By default, selective profiling monitors the hardware events that provide the layer 1 breakdown. Selective profiling only transitions to monitoring layer 2 events if the execution is back-end bound for at least 10% of a time-slice p (100ms by default).

We use 10% as the default threshold, which we empirically determine to be a reasonable threshold (§6.4). We also choose 100ms as a reasonable time-slice for our programs, since the shortest execution across our benchmarks was 1 second and the longest was 2867 seconds. Nonetheless, the percentage and monitoring periods are both configurable. We explore the sensitivity of our results to all these parameters in §6.4.

If selective profiling determines that the execution is also memory bound for at least 10% of the same interval p, it starts monitoring layer 3 events. If selective profiling determines that the execution is L2, L3, or DRAM bound for at least 10% of the same interval p, it transitions to layer 4. Selective profiling then gathers L1, L2, and L3 cache miss events and program locations where the misses occur.

Incremental monitoring is key to ensuring selective profiling's low performance overhead. Successive layers are more costly to monitor as they must count more events—for example, layer 2 requires counting 3× more hardware performance events than layer 1. However, unless selective profiling determines that an execution is back-end bound, it only needs to monitor events at layer 1. As shown in §6.1, only monitoring layer 1 events incurs a negligible overhead (0.7% on average).

Programs can go through phases of different locality issues (*e.g.*, L2 cache misses in one phase and L3 cache misses in another phase). Selective profiling can pinpoint the root cause of the locality problem for each phase, provided the duration of a given phase is at least $4p$ (where p is the duration of selective profiling's time-slice, per layer). If this time-slice is too long, selective profiling may miss some short-running phases. The time slice is configurable. We empirically determine that a time slice of 100ms is effective in practice (§6.4).

3.3 Sampling

In addition to targeted and incremental monitoring, selective profiling also employs sampling at layer 4 for recording L1, L2, and L3 cache misses to further reduce the overhead. Although sampling can reduce run-time overhead, it can also reduce the coverage of data locality issues that selective profiling detects if the sampling period is too high. We define coverage as the ratio of the number of locality issues detected with a given sampling rate to the number of locality issues detected with the highest possible sampling rate.

By default, selective profiling uses a conservative sampling period of 1000 (1 sample per 1000 events), which we have empirically found to yield high coverage (97%, discussed in §6.4) in detecting locality problems across the 13 benchmarks we evaluated. The developer, however, can use a lower sampling period (up to 1 sample per 100 events, as allowed per Linux's `perf` interface). We analyze the coverage versus overhead trade-off of different sampling periods in §6.4.

Selective profiling does not apply sampling in layers 1–3 since sampling reduces coverage. Moreover, in layers 1–3, selective profiling's incremental monitoring reduces the overhead to a negligible amount in all tested applications (on average 1.36%). Therefore, selective profiling does not need to apply sampling at those layers. However, if the overhead of the first three layers is high, selective profiling can optionally enable sampling at those layers as well.

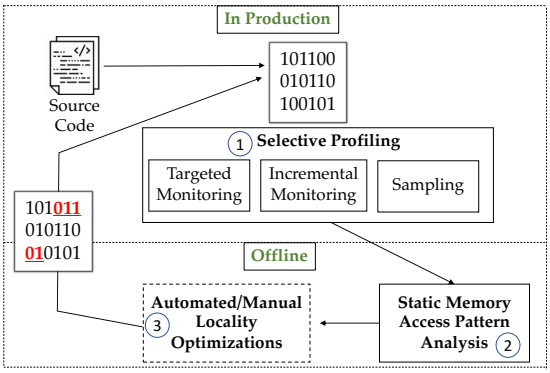

Figure 3: How DMon leverages selective profiling to detect and repair data locality problems.

Now, we describe how data locality information collected via selective profiling can be used to guide automated and manual profile-guided optimizations using DMon.

4 DMon

Selective profiling detects program locations with poor data locality in production. DMon analyzes these locations offline to identify the data access patterns causing data locality issues. Based on the recognized access patterns, DMon applies existing compiler optimizations only to these program locations in a targeted manner to improve data locality. We offer four such optimizations which we describe in §4.2. These optimizations can be automatically applied in most cases for C/C++ applications; for applications written in other programming languages, selective profiling results can still enable manual optimizations (§6.3).

Fig. 3 shows how DMon employs selective profiling to identify and eliminate data locality issues. In step ①, DMon monitors programs in production to determine whether they suffer from poor locality using selective profiling.

Steps ②–③ happen offline, during recompilation. In step ②, DMon determines the memory access patterns that are causing poor data locality (§4.1). In step ③, based on the identified access patterns, either profile-guided automatic optimizations or manual optimizations can be performed to improve data locality (§4.2). The optimized program is then rebuilt and redeployed in production.

4.1 Static Memory Access Pattern Analysis

Once selective profiling identifies memory access instructions that suffer from poor locality in production, DMon analyzes the corresponding program locations offline to determine the cause of the problems. DMon only analyzes memory access instructions that incur more than 10% of the total cache miss events sampled in layer 4 of selective profiling.

Table 1: Four common memory access patterns that cause data locality problems in many applications. Here, we show their examples from the PARSEC [12] benchmark suite.

Benchmark	Code snippet	Access pattern
lu_ncb	`a[i] += alpha*b[i];`	Direct Addressing

radix	`this_key = key_from[i] & bb;`
	`this_key = this_key >> shiftnum;`
	`tmp = rank_ff_mynum[this_key];`
		Indirect Addressing

radiosity	`while(int_list)`
	`{`
	`  if(int_list->dst==inter->dst)return(1);`
	`  int_list = int_list->next ;`
	`}`
		Unbalanced Access

dedup	`if(LstElmnt->seq.l2num > H->Elmnts[Child]->seq.l2num){`	Pointer Chasing

To determine the patterns of data locality issues, we initially analyze the results of selective profiling manually for the benchmarks from the popular PARSEC [12] benchmark suite. Based on our manual analysis of program statements causing data locality issues, we identify four key memory access patterns that can lead to poor data locality. Table 1 shows one example of each of these memory access patterns that cause poor data locality. Perhaps unsurprisingly, all the accesses that contribute significantly to poor data locality are in loops that execute many times and access a relatively large amount of data compared to other memory access operations in the application. These four memory access patterns also cause data locality problems in a diverse set of real-world applications (as we show in §6.3).

For `lu_ncb`, most cache misses that hurt program performance happen while accessing arrays in a loop. Since the loop induction variable (`i`) is directly used to index those arrays, we call this pattern *direct addressing*. For `radix`, the loop induction variable (`i`) is used to index an auxiliary array to load an intermediate value (`this_key`). The loaded intermediate value is used as index while accessing another array, and the last access suffers from poor data locality. We categorize this pattern as *indirect addressing*.

For `radiosity`, most cache misses occur in a while loop, where two member variables (`dst` and `next`) of a structure (`int_list`) are accessed repeatedly. We determine that this structure also contains four other member variables not accessed in this loop. Since only accessing a subset of all member variables causes cache misses, we call this access pattern as *unbalanced access*. Finally, for `dedup`, locality suffers while accessing a chain of structure pointers (pointers `H`, `Elmnts[Child]`, and `seq`, and finally a member variable `l2num`) in a loop. We denote this pattern as *pointer chasing*.

Based on findings of these manual observations, we design the static memory access pattern analysis component of DMon, as shown in Fig. 4. Although DMon's pattern detection is inspired by the manual analysis of locality issues in PARSEC, we show in our evaluation that the patterns DMon identifies generalize to a broad set of systems (§6.2 and §6.3). In particular, the four patterns of poor locality constitute the root causes of all the data locality problems we discover in nine other benchmarks that we had not studied previously.

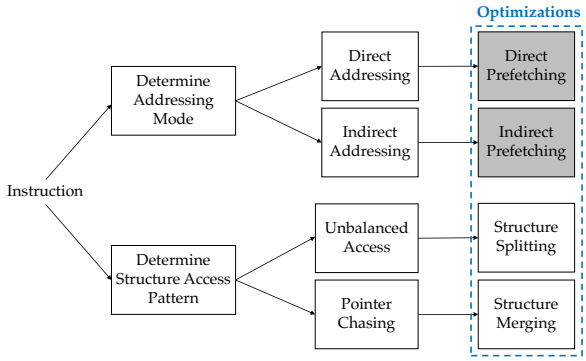

Figure 4: Static memory access pattern analysis in DMon and their corresponding optimizations. Shaded optimizations are mutually exclusive.

As shown in Fig. 4, DMon determines the addressing mode of the memory instruction (*i.e.*, direct or indirect addressing). If the access is made to a structure instance, DMon also determines the type of the access (*i.e.*, unbalanced access and pointer chasing). We discuss each analysis next.

Addressing mode. DMon's static analysis checks if the instruction uses direct or indirect addressing. Here, direct addressing occurs if the computation of the accessed location does not involve another memory address (*e.g.*,`for(i=...) a[i]`). Conversely, indirect memory addressing occurs if the computation of the accessed location involves computing another memory address (*e.g.*,`for(i=...) a[b[i]]`).

Structure access pattern. In addition to determining the addressing mode, DMon's static analysis checks to see if the instruction accesses a member of a structure. DMon does this by mapping the instruction to the compiler intermediate representation and checking if it accesses a structure field. DMon searches for two patterns when a structure member is accessed, namely *unbalanced access* and *pointer chasing*.

DMon concludes that there is an unbalanced access pattern, when accesses to only a *subset* of member variables incur a large fraction of cache misses. Pointer chasing occurs when the accessed memory location belongs to a hierarchy of nested structures (*e.g.*, `A->B->C`).

4.2 Optimizations Implemented in DMon

To show the usefulness of selective profiling, we implement four profile-guided data locality optimization passes using LLVM [56] for C/C++ programs. Our passes optimize the four patterns of poor data locality that DMon identifies. For applications written in other languages, selective profiling results can be used to apply manual optimizations (§6.3).

As shown in Fig. 4, DMon recommends applying a specific optimization technique based on the addressing mode and the structure access patterns of the memory access instruction. While these optimizations are well-known and usually applied statically, selective profiling information enables the targeted

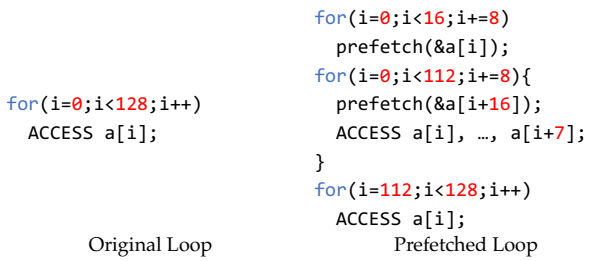

Figure 5: Software prefetching for direct memory access, adapted from [71]. The induction variable is of type `int`. The `prefetch` instruction prefetches one cache line (64 bytes).

application of these optimizations to where they are absolutely needed in a program. As we show in §6.2, DMon-enabled targeted profile-guided optimizations outperform purely static optimizations by 10% on average.

Direct prefetching. The first optimization we implement uses direct prefetching [71] to fix locality problems that stem from memory accesses that use direct addressing. Direct prefetching fetches the cache lines that a program will access in the near future into the cache to improve data locality.

At a high level, direct prefetching works by splitting each loop suffering from poor data locality into three loops, as shown in Fig. 5. The first loop is responsible for prefetching the initial cache line that contains the data accessed by the loop. The second loop starts prefetching the next cache line(s). It also simultaneously performs the original computation that was carried out in the original loop, starting with the first prefetched cache line. The third and last loop completes the computation using the last prefetched cache line.

Direct prefetching can be applied based on compile-time heuristics only. However, this can cause significant performance degradation [29], as we also show in our evaluation (§6.2). This happens because these heuristics might (1) bloat the code footprint by adding unnecessary prefetching instructions (*e.g.*, for lines that would anyways be prefetched by the hardware prefetcher), and (2) cause cache pollution by prefetching data that is not frequently-accessed.

Direct prefetching can also be applied in hardware with popular hardware prefetchers including next-line and stride prefetchers that most modern processors supposedly employ [42, 94]. However, DMon finds that many directly addressed memory accesses suffer from poor data locality, because the underlying hardware prefetchers can not prefetch the cache lines in a timely manner. This is because prefetchers work in a reactive manner, *i.e.*, it takes several iterations for the hardware prefetcher to detect the pattern and start prefetching, but if prefetching is done with explicit instructions, the performance benefits are immediate.

Instead of applying direct prefetching based on compile-time heuristics, our pass only applies it to program locations where DMon identifies that direct addressing access pattern is causing poor data locality.

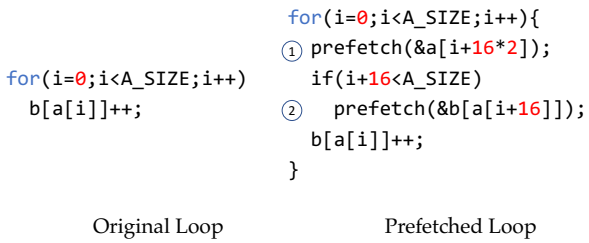

Figure 6: Software prefetching for indirect memory access, adapted from [3].

Indirect prefetching. Our second optimization uses indirect prefetching [3], which is similar to direct prefetching in that it brings data that will soon be used into the cache. Unlike direct prefetching, indirect prefetching also has to prefetch one additional cache line per each level of indirection.

Fig. 6 shows an example of indirect prefetching. Here, the original loop increments elements in an array, `b`. However, the index of the array `b` is computed using another array, `a`. The loop on the right side prefetches the cache line containing the elements of `b` that will be accessed in the near future (prefetch ②). Prefetching the elements of `b` requires accessing the elements of `a`. Thus, to prefetch the elements of `b`, we need to (1) have an array boundary check, and (2) also prefetch the cache line containing the elements of `a` (prefetch ①).

Structure splitting. The third optimization, structure splitting, moves infrequently-accessed members of a structure with a pointer to a new structure that only contains those members. Structure splitting is beneficial only when the total size of infrequently-accessed member(s) is larger than the pointer size. Thus, the size of the original structure is reduced, fitting into fewer cache lines. During memory access pattern analysis, if DMon detects that an unbalanced access pattern (*i.e.*, a subset of structure members are accessed more frequently than others) to members of a structure is causing poor locality, structure splitting is an appropriate optimization.

Fig. 7 shows an example of structure splitting. Here, before structure splitting, the structure `S` has three members (`a`, `b`, `c`) of types `A`, `B`, `C`, respectively. In the original program, an instance of `S` spans two cache lines. Both cache lines need to be accessed each time the program accesses an instance of `S`. For example, if neither of these cache lines is present in the L1 cache, the program will incur two L1 cache misses.

After structure splitting, the new structure `S'` fits in a single cache line (Cache Line 1) because the infrequently-accessed member `b` is moved into a new structure `S2`, residing in its own cache line (Cache Line 2). Consequently, when the program accesses an instance of `S`, it will usually only need to access the cache line (Cache Line 1) containing the frequently-accessed members (`a`, `c`), which would incur a single L1 cache miss (rather than two).

Structure splitting has been previously explored [22] in type-safe languages (*e.g.*, Java). However, implementing structure splitting in a type-unsafe language (we target C/C++) is more challenging. This is because structure splitting needs to

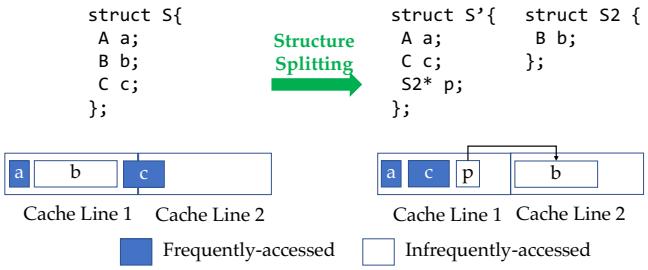

Figure 7: Structure splitting, example adapted from [22].

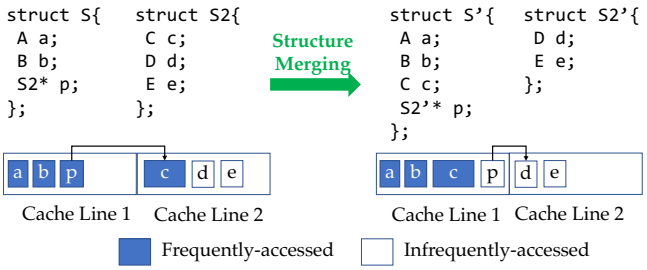

Figure 8: Structure merging example.

ensure that the program continues operating correctly when the layout of the structure is modified. More specifically, all the instructions that used to refer to the old layout need to be updated to refer to the new layout.

In our optimization pass, we addresses this challenge using a complete, interprocedural, inclusion-based pointer analysis [5] that can determine all instructions that could possibly access the split structures. As shown in §6.2, this optimization can automatically be applied in all but one of the benchmarks.

Structure merging. The final optimization, structure merging, is the inverse of structure splitting as it replaces a frequently-accessed pointer member of a structure with the data that the pointer references. The key idea is to eliminate the pointer chasing pattern that DMon identifies by removing a level of indirection for frequently-accessed elements.

Fig. 8 shows an example of structure merging. Before merging, the structure `S` has three members (`a, b, p`) of types `A, B, S2*`, respectively. The instance of `S` resides in the first cache line, and the pointer `p` points to an instance of structure `S2` that resides in the second cache line. The size of `a`, `b`, and `c` is such that they can all fit in one cache line. If `c` is accessed as frequently as `a` and `b`, then data locality can be improved by merging these two structures into one. This structure merge will also bypass one memory access (`S'->C` instead of `S->S2->C`). Structure merging only combines member variables across different structure types and hence does not perform exhaustive data structure conversions (*e.g.*, transforming a linked list into an array) [22, 23].

DMon employs structure merging conservatively so that it will only be applied if soundness can be guaranteed. In other words, DMon applies this optimization only if all updates via the structure pointer can be safely redirected (*e.g.*, in Fig. 8, all changes to `S->S2->C` could be replaced by `S'->C`). To ensure this, structure merging also uses the same pointer analysis [5] that structure splitting uses.

Other optimizations.. DMon can be easily extended to accommodate additional optimizations if needed to fix different patterns of memory accesses which cause data locality problems. For example, DMon can work as a framework to apply optimizations like loop reordering, blocking, tiling, and strip mining in a profile-guided manner. However, many of these optimizations require expensive memory access trace collection which can not be deployed in production due to high overheads [64]. In the future, we intend to explore how these optimizations can be applied based on more efficient profiling.

5 Implementation

DMon's selective profiling prototype is implemented for Intel processors. In particular, selective profiling relies on the Linux `perf` [97] interface for profiling hardware events in layers 1–4 (§3). We initially build the benchmarks using debug information and the highest level of compiler optimization (`-O3`), and then use the `strip` utility [101] to remove the debug information. During in-production monitoring, selective profiling records the program counter for each sampled cache miss event in layer 4. To efficiently deal with multi-threaded applications, selective profiling maintains a per-thread buffer (2MB per thread) to record the program counters. When the buffer gets full, the previous samples get overwritten. Offline, DMon uses the program counter, the stripped debug information, and the program binary to find the source code location where a cache miss occurred in production.

We implement DMon's optimizations in the LLVM [56] compiler framework. We use `clang` [99] to generate the LLVM intermediate representation (IR) that the optimization passes of DMon can operate on. The optimizations rely on the program's debug information to map the source code location to LLVM IR, because a 1-to-1 mapping between machine code and LLVM IR does not exist.

Similar to other state-of-the-art profile-guided optimization techniques [17, 68], DMon's use of debug information for mapping machine code to LLVM and locating code locations to optimize can introduce inaccuracies. This happens due to optimizations such as inlining. Although it is possible to improve the accuracy of such mapping using more invasive instrumentation and tracing [7], this would be prohibitively costly for production usage [48]. In our evaluation (§6), we show that the accuracy provided by debug information can lead to substantial speedup.

The optimizations for structure splitting and structure merging use a whole-program pointer analysis [19].

6 Evaluation

In this section, we first evaluate the efficiency of selective profiling by measuring its run-time monitoring overhead. Then, we evaluate the effectiveness of DMon by showing the extent to which fixing the locality problems detected by

DMon improves performance of popular benchmarks. Next, we evaluate selective profiling's generality by applying it to widely-used real-world applications. Finally, we perform sensitivity studies to evaluate how DMon's overhead and detection results vary in response to changes of the different system parameters of DMon.

Software. All experiments are conducted in Ubuntu 18.04 (kernel version 4.15.0-46-generic). The static compiler analyses are implemented in LLVM (7.0.0) on bitcode emitted by clang. Therefore, we use `clang 7` as the baseline compiler.

Hardware. We use a 20-core 2.2 GHz Intel Xeon NUMA (with 2 sockets) machine, with 64 KB of L1-cache (32 KB instruction and 32 KB data), 1024 KB of L2-cache, 14 MB of L3-cache (shared across the same NUMA node), and 96 GB of RAM. Like most Intel processors, each core in the machine uses two hardware prefetchers (next-line and sequential load history driven prefetchers) in the L1 data cache and two hardware prefetchers (adjacent cache line and streaming prefetchers) in the L2 cache [42, 94]. We configure multi-threaded applications and benchmarks to run with 8 threads.

Benchmarks. We use a combination of benchmarks and real-world programs that have been widely used in prior performance profiling and optimization work. In particular, we use all 12 benchmarks from the PARSEC [12] suite, all 11 benchmarks from the SPLASH-2X [103] suite, and all 3 benchmarks written in C from the NPB [10] suite, as well as `HashJoin`, `RandomAccess`, `kcstashtest`, and `DIS`, which are programs with poor data locality from other popular benchmark suites [11, 24, 63, 73]. We also study one of the most popular and heavily-optimized open-source databases, `PostgreSQL` [81], running the TPC-H analytical workload [26]. Finally, we study real-world applications from the Renaissance benchmark suite [83].

Metrics. In all our plots, we report speedup numbers as the ratio between the execution time of the original application compiled with the highest level of optimization (`-O3`) and its run time after applying DMon-guided optimizations. Negative speedup denotes slow-down. Similarly, we report selective profiling overhead as the percentage increase in benchmark execution time while enabling selective profiling. We report performance data as the average of 25 runs in all experiments.

6.1 Selective Profiling Efficiency

We evaluate the selective profiling efficiency by studying the overhead selective profiling incurs during dynamic detection of locality problems. Fig. 9 shows this overhead. We present results for all the benchmarks we evaluated, including the ones for which selective profiling did not find locality optimization opportunities. For each benchmark, we present the overhead of each layer of monitoring (1–4) that selective profiling employs. Since, selective profiling monitors only one layer at a time, the effective overhead for a given program is less than the maximum overhead across four layers.

Across all layers and benchmarks, selective profiling incurs up to 4.92% overhead, and on average only 1.36% overhead. On average, selective profiling incurs an overhead of 0.7% in layer 1, an overhead of 1.5% in layer 2, an overhead of 2.5% in layer 3, and an overhead of 2% in layer 4. For benchmarks that do not have locality problems, layers 2–4 are never triggered.

In only 3 out of all 28 benchmarks, selective profiling incurs more than 3% overhead: `IS` (4.6%), `kcstashtest` (4.2%), and `HashJoin` (4.9%). However, as we detail in §6.2, optimizations suggested by DMon also provide greater speedups for these benchmarks than for others (`IS` 30.3%, `kcstashtest` 32.4%, and `HashJoin` 53.1%—compared to 16.83% average speedup enabled by DMon). These benchmarks suffer the most from poor locality, and consequently, selective profiling incurs more overhead to pinpoint the root cause of those problems.

6.2 Effectiveness

We evaluate the effectiveness of DMon by studying (1) data locality problems detected by DMon, (2) speedups provided by DMon-guided optimizations, (3) comparison of the speedups provided by DMon-guided optimizations to the speedups provided by Google's AutoFDO [17]—the state-of-the-art profile-guided locality optimization approach, (4) whether DMon-guided optimizations generalize across different program inputs, and (5) the overhead on compilation times due to DMon-guided optimizations.

Locality issues detected by DMon. Table 2 summarizes the data locality problems that DMon detects. For brevity, Table 2 omits benchmarks where less than 10% of the execution time is bounded by locality problems, as these benchmarks could not benefit from eliminating locality improvements. We also omit these benchmarks in our average performance numbers.

Additionally, Table 2 shows the most prominent level of the memory hierarchy for the locality issues detected by selective profiling. Note that, in many cases, DRAM accesses constitute the locality bottlenecks. This is expected, since the highest-latency memory access instructions are served from DRAM. Finally, Table 2 also reports the program locations (as "file": "line number") that suffer the most from poor locality, along with the optimizations DMon recommends in each case.

As shown, DMon successfully identifies locality problems and suggests appropriate optimizations in each case. In all cases but one (`fmm`), DMon applies optimizations automatically. For `fmm`, while the direct prefetching is applied automatically, structure splitting cannot be applied automatically. This is because, due to excessive type casts, the compile-time optimization cannot exactly determine which program statements may access the modified structure, and therefore cannot automatically update such statements. Nonetheless, since DMon points the developer to the exact source of the locality issue in `fmm`, the fix can easily be applied manually with an 8 LOC update. Moreover, structure splitting and merging can be applied automatically for other applications (`dedup` and `radiosity`)

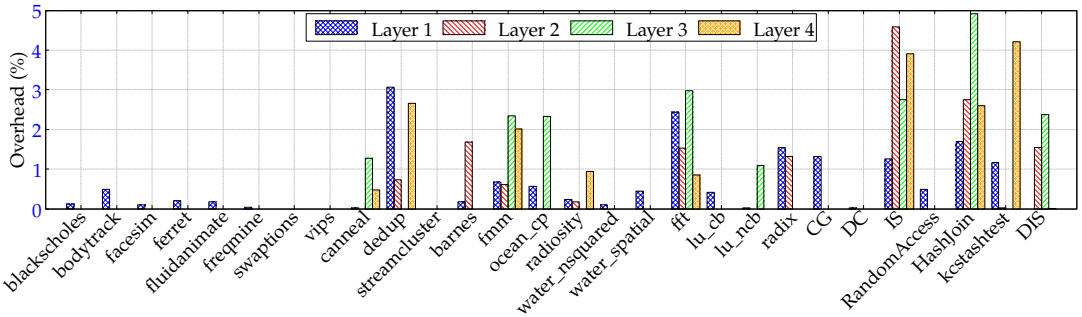

Figure 9: Monitoring overhead of selective profiling (All $\sigma < 0.02\mu$).

Table 2: DMon's detection results of locality problems.

Benchmark	Execution time (seconds)	Memory hierarchy bottleneck	Program location	Optimization	Automated fix?
canneal	71.8	L3, DRAM	netlist_elem.cpp: 80	Direct Prefetching	Yes
dedup	5.1	DRAM	binheap.c: 93	Structure Merging	Yes
fmm	18.8	DRAM	interactions.C: 169	Structure Splitting	No
				Direct Prefetching	Yes
ocean_cp	36.2	L2, L3, DRAM	multi.C: 273	Direct Prefetching	Yes
radiosity	95.8	L2, L3	rad_tools.C: 399	Structure Splitting	Yes
fft	1.2	DRAM	fft.C: 765	Direct Prefetching	Yes
lu_ncb	47.8	L3, DRAM	lu.C: 466	Direct Prefetching	Yes
radix	6.1	L2, L3, DRAM	radix.C: 624	Indirect Prefetching	Yes
IS	1	L3, DRAM	is.c: 392	Indirect Prefetching	Yes
RandomAccess	607.1	DRAM	randacc.c: 125	Indirect Prefetching	Yes
HashJoin	2867.3	L3, DRAM	npj2epb.c: 300	Indirect Prefetching	Yes
kcstashtest	3.20	L2, L3, DRAM	kcstashdb.h: 146	Direct Prefetching	Yes
DIS	165.3	L2, L3, DRAM	transitive.c: 107	Direct Prefetching	Yes

Table 3: Speedup comparison between DMon and compile-time optimizations.

Benchmark	Speedup provided by compile-time optimizations (%)	Speedup provided by DMon (%)
canneal	-7.90	1.07
dedup	-18.90	3.65
fmm	2.83	2.68
ocean_cp	-1.06	2.90
radiosity	-7.14	11.21
fft	1.11	4.57
lu_ncb	3.49	19.40
radix	0.96	1.85
IS	30.52	30.29
RandomAccess	38.83	47.67
HashJoin	9.74	53.14
kcstashtest	37.41	32.39
DIS	-0.28	7.93

where the automatic transformation can identify and update all statements pointing to the split and merged structures.

Speedup. Table 3 compares the speedup provided by the DMon-guided optimizations. Optimizations guided by DMon provide up to 53.14% and on average 16.83% (8% median) speedup. To study the impact of the targeted optimizations guided by selective profiling results, we also report the speedup achieved by the same optimizations if they are applied indiscriminately (*i.e.*, in a non-targeted way), through purely-static compiler passes [3, 71].

As shown in Table 3, DMon-guided optimizations outperform compile-time optimizations in 10/13 benchmarks. Crucially, static optimizations hurt performance in 5/13 cases due to being applied too broadly (with no runtime information), and therefore causing outcomes such as cache pollution and code bloat. DMon-guided optimizations always improve the performance. In 3/13 benchmarks where static optimizations outperform DMon-guided optimizations, the margin is $\leq 5\%$ which can be reduced by reducing the incremental monitoring threshold (default, 10%) of selective profiling.

Comparison against Google AutoFDO. We compare the speedup provided by DMon-guided optimizations to that of Google's AutoFDO [17], the state-of-the-art profile guided

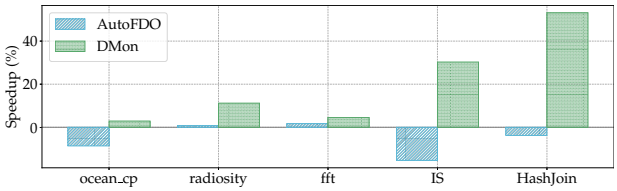

Figure 10: Speedup comparison to AutoFDO (All $\sigma < 0.09\mu$)

optimization technique. AutoFDO has limited data locality optimization capabilities [68]; our comparison is thus limited to five benchmarks for which AutoFDO can optimize locality.

We compare the speedup provided by DMon-guided optimizations to the speedup provided by AutoFDO in Fig. 10. As shown, DMon-guided optimizations provide better speedup than AutoFDO for all five benchmarks. This is because AutoFDO could only identify data locality problems that can be solved by performing direct prefetching optimizations. By contrast, DMon can identify other data locality issues that can be addressed by additional locality optimizations (*i.e.*, indirect prefetching, structure splitting, and structure merging).

For example, AutoFDO's direct prefetching slows down the execution of `IS` by 15%, while DMon-guided indirect prefetching provides a 30% speedup. Even for cases where both DMon

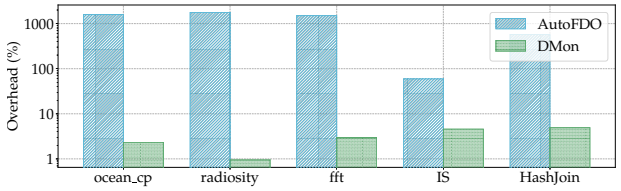

Figure 11: Overhead comparison to AutoFDO (All $\sigma < 0.07\mu$)

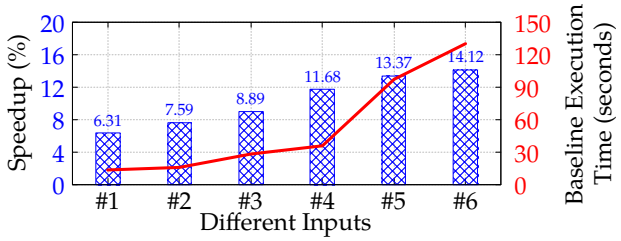

Figure 12: DMon-generated optimization after observing input #4 generalizes to unseen inputs (All $\sigma < 0.01\mu$).

and AutoFDO suggest direct prefetching (*e.g.*, `ocean_cp`), DMon-guided optimizations outperform AutoFDO, because, unlike AutoFDO, DMon provides hints as to where (*e.g.*, L1, L2, or L3) the cache line should be prefetched.

We compare selective profiling overhead against AutoFDO's profiling overheads in Fig. 11. For the 5 benchmarks in this study, selective profiling incurs 3.3% mean overhead, whereas AutoFDO incurs 978% mean overhead, making the latter unsuitable for production use.

Generalization across program inputs. Profile-guided optimizations perform best when the application is optimized with a profile that is representative of the application's common behavior [17, 79, 95]. DMon-guided fixes also generalize if the program shows similar data locality behavior across different inputs. Therefore, we evaluate DMon's generality across different program inputs for 9 benchmarks. These program inputs vary widely both in terms of input size (from megabytes to gigabytes) as well as execution times needed to process the input (from seconds to minutes).

We report a detailed case study using the `radiosity` benchmark to determine how well the locality optimizations suggested by DMon generalize to different inputs. We choose this benchmark because the fix suggested by DMon is structure splitting—an optimization that modifies the data layout, and hence has the potential to be affected by changing program inputs. Fig. 12 shows the speedup provided by DMon-guided optimizations for `radiosity` for various input sizes.

Here, for brevity, we refer to different input sizes using "#1" through "#6". DMon only observes the execution for the randomly selected input #4. After observing input #4, DMon-guided optimizations are applied. Then, all inputs are rerun with the newly-optimized program, with the results of this run reported in Fig. 12. As shown, the optimization suggested by DMon generalizes well to other inputs, providing considerable

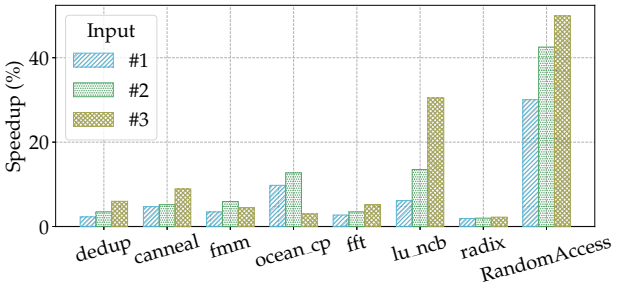

Figure 13: Input generalization (All $\sigma < 0.04\mu$)

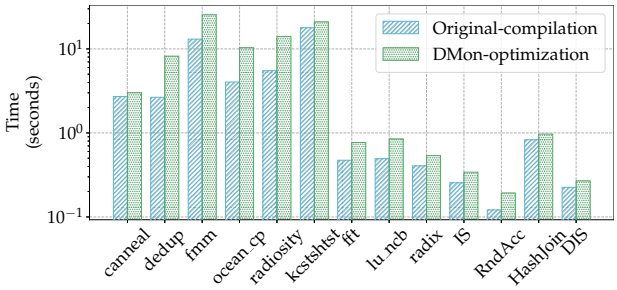

Figure 14: Overhead of DMon-guided optimizations compared to baseline compilation time ($\sigma < 0.1\mu$, log-scaled y).

speedups in each case. Longer executions that use *larger* inputs benefit more from optimizations.

Fig. 13 shows how DMon-guided optimizations improve data locality for unobserved inputs of several other benchmarks. Here, we include all benchmarks with at least 3 inputs. Across all evaluation targets, we find that data locality behavior follows a similar trend for different inputs. Hence, DMon's fixes generalize to different inputs for these benchmarks.

Recompilation overhead. We evaluate the offline recompilation overhead while applying DMon-guided optimizations, though this does not impact the production overhead. We perform this experiment, because automated structure splitting and merging require pointer analysis, which is known to be expensive [55]. However, the specific pointer analysis we employ is flow- and context- insensitive and scales well [40].

Fig. 14 shows the offline compilation overhead incurred by our DMon-guided optimizations on top of the baseline compilation overhead (`clang`). On average, DMon-guided optimizations incur 72% more overhead. However, the optimization takes on average less than 7 seconds and is no longer than 26 seconds. Even for large applications (*e.g.*, PostgreSQL [92] code base has over 1M LOC), the analysis takes 307 seconds. For an offline process, we believe these durations are reasonable and on par with standard compiler transformations that use whole-program pointer analysis. Moreover, this is a one-time compile-time overhead and will be amortized for long-running applications (*e.g.*, data-center applications that are compiled once but run on thousands of servers for days). Finally, structure splitting and merging can be applied manually if the cost of pointer analysis is deemed prohibitive.

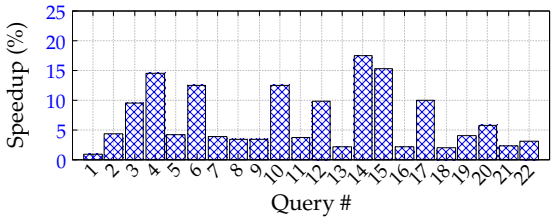

Figure 15: Speedup due to DMon-guided optimizations for 22 TPC-Hqueries on `PostgreSQL` (All $\sigma < 4.53\%$ of μ).

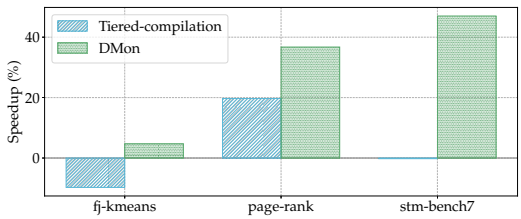

Figure 16: Speedup provided by selective profile-guided optimizations for just-in-time (JIT) compiled applications against tiered compilation (All $\sigma < 7.68\%$ of μ).

6.3 Real-World Case Studies

We evaluate the applicability of selective profiling and DMon to large systems by studying (1) speedups provided by DMon-guided optimizations on PostgreSQL [81]—one of the most popular database systems, and (2) speedups achieved after manual repair of data locality problems detected by selective profiling for just-in-time (JIT) compiled real-world applications from the Renaissance benchmark suite [83].

PostgreSQL case study. We evaluate DMon's ability to improve the locality (and thereby performance) of `PostgreSQL` v11.2 [81], one of the most popular open-source database management systems. For this study, we run the popular TPC-H [26] queries on a 1GB database stored in PostgreSQL. We intentionally select the database size to fit in memory to ensure a memory-bound workload (instead of disk-bound one), as the vast majority of real-world databases fit in memory [67, 80].

To evaluate DMon, we profile PostgreSQL with DMon while serving all 22 TPC-H queries. For these queries, selective profiling incurs 1.2% average and 2.7% maximum overhead. For `PostgreSQL`, DMon identifies a locality problem in a function (`ExecParallelHashNextTuple`) that accesses the members `area` and `parallel_state` of structure `hashtable` [39]. DMon identifies that this memory access is the primary reason for poor data locality in 6 out of 22 TPC-H queries. Moreover, this memory access causes L2 and L3 cache misses for all 22 TPC-H queries. The cause of the locality problem in this case is pointer chasing. Structure merging automatically repairs this problem and speeds up all 22 TPC-H queries, as shown in Fig. 15. The L3 cache misses in `PostgreSQL` are reduced by up to 22.11% (3.05% on average) and the latency of the 22 TPC-H queries are improved by up to 17.48% (6.64% on average). We also test optimized `PostgreSQL` based on DMon-profile on larger databases (10 and 100GB), where DMon improves the latency of the 22 TPC-H queries by 4.68% on average. For larger databases (10 and 100GB), the overall performance gain due to DMon's optimizations are comparatively less than (2% on average) that of smaller databases (1GB). That is because the performance of `PostgreSQL` for larger databases are primarily bottlenecked by storage I/O costs.

These results are particularly encouraging, considering that `PostgreSQL` is one of the most heavily-optimized codebases, having been improved by developers over the past 20 years. Most database developers hand-tune their code using the TPC benchmarks as regression tests (*i.e.*, their performance is best on TPC). This fact makes it even more promising that DMon-guided optimizations are able to improve the performance of these benchmark queries on a mature database system. We reported this data locality issue to the developers of `PostgreSQL` (for the version 11.2), which they have fixed since then.

Renaissance case study. A key advantage of just-in-time (JIT) compilation over ahead-of-time compilation (*e.g.*, Java vs. C++) is that JIT can apply dynamic optimizations—including limited data locality optimizations—using tiered compilation [65]. We compare selective profile-guided data locality optimizations to tiered compilation from OpenJDK [100] on real-world applications from the Renaissance suite [83]. For these applications, selective profiling incurs 2.2% average and 2.6% maximum overhead.

We use selective profiling to detect data locality issues in three Renaissance applications (jdk-concurrent `fj-kmeans`, apache-spark `page-rank`, and Scala `stm-bench7`). We omit other Renaissance benchmarks for which selective profiling does not find any data locality problems. Most of the data locality issues found here corresponds to Java/Scala source code (we map binary instruction information back to Java code using perf-map-agent [45]) of Renaissance applications. Since currently DMon's optimizations only support C/C++ applications, we manually apply data locality optimizations to these applications. In all cases, we modify <10 LOC.

As shown in Fig. 16, selective profile-guided optimizations provide on average 26% and up to 47% more speedup than tiered compilation. This demonstrates that selective profiling is effective even for JIT-compiled applications.

Apart from these real-world case studies, we have also tested DMon on `Memcached` [35] and `RocksDB` [33] with YCSB benchmarks [25]. For these two applications, the individual pieces that make up the locality issues are relatively minor. Compiler-based data locality optimizations typically add extra instructions and logic in the code, which only helps when there are many cache misses causing slowdowns. For program statements responsible for a relatively small percentage of all cache misses (less than 5%), applying these optimizations do not provide any speedup, as the extra code and logic outweighs the benefits.

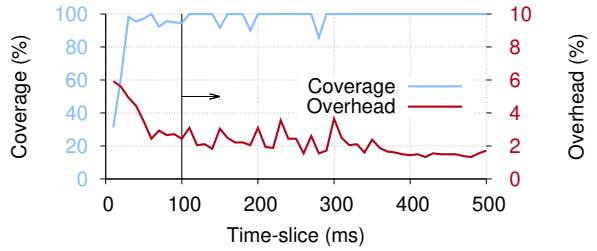

Figure 17: Effect of granularity of in-production time-slice on detection coverage and overhead (All $\sigma < 3.03\%$ of μ).

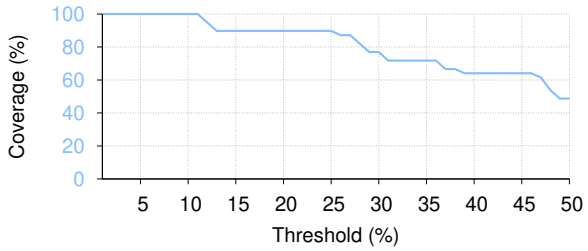

Figure 18: Effect of incremental monitoring threshold on the coverage of locality problems selective profiling detects across all benchmarks.

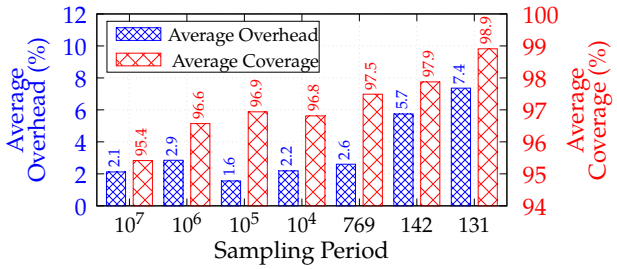

Figure 19: Effect of sampling period on the coverage of locality problems selective profiling detects and the average overhead across all benchmarks ($\sigma < 0.01\mu$).

6.4 Sensitivity Analysis

We evaluate the impact of selective profiling's different parameters on effectiveness (coverage) and efficiency.

In-Production Monitoring Time-Slice. The granularity of the monitoring time-slice is a key design decision for selective profiling's incremental monitoring scheme (§3). Small time-slices allow selective profiling to identify locality problems for shorter-running applications, but also trigger frequent transitions during incremental monitoring and result in higher monitoring overhead. On the other hand, larger time-slices lower overhead but may fail to detect locality problems for shorter-running programs.

Fig. 17 shows the impact of the time-slice granularity on selective profiling's detection coverage (left y-axis) and overhead (right y-axis) for the benchmark, (`lu_ncb`). We vary the time-slice granularity from 10ms to 500ms (with 10ms increments) and measure selective profiling's coverage in detecting data locality issues and the associated performance overhead.

As shown in Fig. 17, selective profiling has lower coverage and higher overhead for smaller time-slices. As the time-slice granularity increases, selective profiling achieves greater coverage with lower overhead. Selective profiling's coverage is lower for smaller time-slices because selective profiling cannot monitor sufficient performance events in a small time slice. Beyond 100ms, both the coverage (99.07% on average with standard deviation of 3%) and the overhead (2.04% on average with standard deviation of 0.6%) lines flatten. Ergo, we set selective profiling's default time-slice as 100ms.

Incremental Monitoring Threshold. We vary the threshold of incremental monitoring (§3) from 1% to 50% and measure the coverage of data locality issues selective profiling detects for all 13 benchmarks in Table 2. 100% coverage is achieved when there is no incremental monitoring (*i.e.*, DMon continuously monitors events at the all levels of the locality tree). As shown in Fig. 18, selective profiling achieves greater than 80% coverage if the incremental monitoring scheme uses a threshold of <29%. Nevertheless, we set the default-threshold as 10%, as this threshold achieves 100% coverage.

In-Production Sampling Period. As described in §3, sampling period is a key design decision for selective profiling. Fig. 19 shows the impact of the sampling period on the coverage of locality issues selective profiling detects and its runtime overhead. We compute coverage with respect to the baseline coverage of 100%, achievable via the lowest possible sampling period offered by Linux `perf` (sampling every 100th event). A sampling period k on the x-axis means selective profiling will record one out of each k events. The left y-axis represents the runtime overhead and the right y-axis represents the coverage of locality issues selective profiling detects.

The overhead and coverage reported in Fig. 19 are arithmetic averages over all benchmarks. A smaller sampling period increases the overhead of selective profiling, but also increases coverage. In our experiments, we chose a sampling period of 1000, which yields a high coverage of 97% with 2.6% overhead on average in layer 4 of selective profiling.

7 Related Work

DMon finds data locality problems with low overhead using selective profiling, identifies the root cause behind the problem, and guides optimizations to eliminate the problem. Existing profilers are not able to determine the root causes of data locality problems without incurring a high overhead.

Profilers. General-purpose profilers [57, 97, 102] report program hotspots without identifying the root cause behind performance problem. Consequently, recent studies propose specialized profilers to locate root cause for specific performance issues. Parallel profilers [36, 41, 44, 46] focus on critical path profiling to estimate potential performance gain [28, 107]. Synchronization profilers [4, 30, 108, 110] identify lock contention. Similarly, we design selective profiling as a special-

ized profiling technique for data locality. Selective profiling uses the APIs of a state-of-the-art profiler, Linux `perf`, and targets a subset of the events explored as part of the Top-Down [106]. Our main contributions over `perf` and Top-Down are: (1) full automation in profiling, (2) low-enough overhead for production deployment, (3) ability to automatically identify targeted optimizations based on the underlying performance problem.

Profile-guided data locality optimizations. Profile-guided approaches collect execution traces to identify where optimizations can be applied [21, 49, 51, 52, 59, 60, 69, 78]. State-of-the-art techniques [17, 37, 74–76] primarily address instruction locality. While prior work [50, 53, 86] also optimizes data locality, these solutions incur >10% profiling overhead. Selective profiling, however, incurs only 1.36% overhead on average (§6.1).

Static locality optimizations. Static approaches use complex analysis techniques to find opportunities to apply locality-improving transformations [14, 16, 18, 31, 47, 58, 66, 88, 105]. Alas, these techniques use compile-time heuristics to apply transformations, which can lead to sub-optimal speedups or even reductions in performance. To avoid these issues, we use application profiles collected by selective profiling to apply optimizations in a *targeted* manner, leading to better speedups and avoiding transformations which hurt performance.

Dynamic locality optimizations. There are several proposals for monitoring program execution and modifying program binaries to improve locality on the fly [32, 72, 89, 96]. These techniques require non-existent hardware support and incur high overhead (up to 6× [96]). Just-in-time (JIT) compilation techniques [21, 43] provide limited data locality optimizations. On the other hand, DMon works with existing hardware, incurs negligible overhead, and guides optimizations that provide better speedup (16.83% on average).

8 Conclusion

Poor data locality is a major performance problem that hurt applications in production. Unfortunately, existing data locality profilers are not efficient enough to be deployed in production. This is limiting, since production profiles are difficult to replicate offline. We address this problem by selective profiling, a technique capable of discovering data locality problems with negligible overhead (on average 1.36%) in production. We also design DMon, which guides automatic and manual data locality optimizations based on profiles generated using selective profiling. For an extensive set of real-world applications and widely-used benchmarks, DMon provides up to 53.14% and on average 16.83% speedup for the cases where DMon applies targeted optimizations after detecting significant data locality problems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Michael Stumm, for their insightful feedback and suggestions. This work was supported by the Intel Corporation, the NSF grants #1553169, #1629397, #2010810, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank Yifan Zhao for running several `PostgreSQL` experiments. We also thank Xiaohe Cheng, Zhiqi Chen, and Shariq Hafeez for testing DMon on various applications. Finally, we thank Kevin Loughlin for his feedback on this paper's earlier versions.

A Artifact Appendix

Abstract

We provide the open-source public repository as an artifact for DMon.

Scope

This artifact allows to validate the effectiveness and efficiency of the selective profiling technique.

Contents

This artifact includes one end-to-end example of how to apply selective profiling to monitor in-production data locality issues and one example of data locality optimization applied in a targeted manner based on the output of selective profiling.

Hosting

We host the artifact on Github. Our open-source artifact repository can be obtained from `https://github.com/efeslab/DMon-AE`. The branch name for the artifact is `main`. The commit hash for the artifact is `d9a0f31`.

Requirements

Intel processor, Linux `perf`, `pmu-tools` [54] that implement the Top-Down methodology [106], and LLVM [56].

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 305–317, Piscataway, NJ, USA, 2017. IEEE Press.

[4] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313, 2017.

[5] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[6] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[7] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

[8] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 462–473. ACM, 2019.

[9] Reza Azimi, Michael Stumm, and Robert W Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, 2005.

[10] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[12] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56. IEEE, 2008.

[13] Michael D Bond and Kathryn S McKinley. Continuous path and edge profiling. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140. IEEE Computer Society, 2005.

[14] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, volume 43, pages 101–113. ACM, 2008.

[15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[16] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 252–262, New York, NY, USA, 1994. ACM.

[17] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23. ACM, 2016.

[18] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. Locality analysis through static parallel sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 557–570, 2018.

[19] Jia Chen. Andersen's inclusion-based pointer analysis re-implementation in LLVM. https://github.com/grievejia/andersen, 2018. [Online; accessed 16-Nov-2018].

[20] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.*, 42(9):1045–1057, September 1993.

[21] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization.

In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 332–340, New York, NY, USA, 2006. ACM.

[22] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[23] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[24] cloudfare. kyotocabinet/kcstashtest.cc at master - cloudflare/kyotocabinet. https://github.com/cloudflare/kyotocabinet/blob/master/kcstashtest.cc, 2013. [Online; accessed 4-April-2019].

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[26] Transaction Processing Performance Council. Tpc-h. [Online; accessed 23-April-2019].

[27] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 219–228, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[29] Daniel Lemire. Is software prefetching (__builtin_prefetch) useful for performance?, 2018. [Online; accessed 24-April-2019].

[30] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.

[31] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Acm Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.

[32] Tyler Dwyer and Alexandra Fedorova. On instruction organization. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[33] Facebook. Rocksdb: A persistent key-value store for flash and ram storage. https://github.com/facebook/rocksdb/, 2021.

[34] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[35] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.

[36] Saturnino Garcia, Donghwan Jeon, Christopher M Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. *ACM SIGPLAN Notices*, 46(6):458–469, 2011.

[37] Google. Propeller: Profile guided optimizing large scale llvm-based relinker. https://github.com/google/llvm-propeller, 2020.

[38] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[39] The PostgreSQL Global Development Group. Line number 3225. https://github.com/postgres/postgres/blob/master/src/backend/executor/nodeHash.c.

[40] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[41] Yuxiong He, Charles E Leiserson, and William M Leiserson. The cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, 2010.

[42] Ravi Hegde. Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers. *Intel Software Network*, 2008. [Online; accessed 5-December-2020].

[43] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented*

Programming, Systems, Languages, and Applications, OOPSLA '04, pages 69–80, New York, NY, USA, 2004. ACM.

[44] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck identification and scheduling in multithreaded applications. *ACM SIGARCH Computer Architecture News*, 40(1):223–234, 2012.

[45] jvm-profiling-tools. perf-map-agent, 2018. [Online; accessed 6-December-2020].

[46] Melanie Kambadur, Kui Tang, and Martha A Kim. Harmony: Collection and analysis of parallel block vectors. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–463. IEEE, 2012.

[47] Mahmut Taylan Kandemir. A compiler technique for improving whole-program locality. In *ACM SIGPLAN Notices*, volume 36, pages 179–192. ACM, 2001.

[48] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[49] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient tracing of cold code via bias-free sampling. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 243–254, 2014.

[50] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*, pages 101–110. IEEE, 2014.

[51] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[52] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, ISCA 2021, June 2021.

[53] Tanvir Ahmed Khan, Yifan Zhao, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Huron: hybrid false sharing detection and repair. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 453–468, 2019.

[54] Andi Kleen. Github - andikleen/pmu-tools: Intel pmu profiling tools. https://github.com/andikleen/pmu-tools.

[55] William Landi and Barbara G Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, 1991.

[56] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, pages 1–2, 2008.

[57] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[58] Jonathan Lifflander and Sriram Krishnamoorthy. Cache locality optimization for recursive programs. In *ACM SIGPLAN Notices*, volume 52, pages 1–16. ACM, 2017.

[59] Xu Liu and John Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 171–180. IEEE Computer Society, 2011.

[60] Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[61] Xu Liu, Kamal Sharma, and John Mellor-Crummey. Arraytool: a lightweight profiler to guide array regrouping. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 405–415. IEEE, 2014.

[62] Xu Liu and Bo Wu. Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 47:1–47:12, New York, NY, USA, 2015. ACM.

[63] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, volume 213. Citeseer, 2006.

[64] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 185–195, 2018.

[65] Markus Weninger. What exactly does -xx:-tieredcompilation do?, 2016. [Online; accessed 11-November-2019].

[66] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996.*, pages 94–104, 1996.

[67] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

[68] Mircea Trofin. Support for cache prefetching profiles. by mtrofin · pull request #75 · google/autofdo, 2018. [Online; accessed 17-November-2019].

[69] Svetozar Miucin and Alexandra Fedorova. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems*, pages 243–253. ACM, 2018.

[70] Todd C Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, to the Department of Electrical Engineering.Stanford University, 1994.

[71] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, New York, NY, USA, 1992. ACM.

[72] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2018.

[73] Joseph Musmanno. Data intensive systems (dis) benchmark performance summary. Technical report, TITAN SYSTEMS CORP WALTHAM MA, 2003.

[74] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165. ACM, 2018.

[75] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14. IEEE Press, 2019.

[76] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 119–130, 2021.

[77] Paratools. Threadspotter. http://threadspotter.paratools.com/, 2019. [Online; accessed 22-Oct-2019].

[78] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, pages 335–348. ACM, 2010.

[79] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM.

[80] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[81] PostgreSQL. Postgresql: The world's most advanced open source relational database. [Online; accessed 23-April-2019].

[82] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[83] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, Doug Simon, et al. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47. ACM, 2019.

[84] Manman Ren and Shane Nay. Improving iOS Startup Performance with Binary Layout Optimizations, 2019. [Online; accessed 25-Oct-2019].

[85] Roman Oderov. Sampling and vtune's disadvantages, 2012. [Online; accessed 23-April-2019].

[86] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[87] J Sedlacek and H Thomas. Visualvm all-in-one java troubleshooting tool, 2018.

[88] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices*, 34(5):215–228, 1999.

[89] Jithendra Srinivas, Wei Ding, and Mahmut Kandemir. Reactive tiling. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 91–102. IEEE, 2015.

[90] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[91] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[92] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM.

[93] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[94] Vish Viswanathan. Disclosure of hardware prefetcher control on some intel processors. *Intel SW Developer Zone*, 2014.

[95] David W Wall. Predicting program behavior using real or estimated profiles. *ACM SIGPLAN Notices*, 26(6):59–70, 1991.

[96] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. On-the-fly structure splitting for heap objects. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2012.

[97] Wikipedia contributors. Perf (linux) — Wikipedia, the free encyclopedia, 2018. [Online; accessed 24-April-2019].

[98] Wikipedia contributors. Vtune — Wikipedia, the free encyclopedia, 2018. [Online; accessed 23-April-2019].

[99] Wikipedia contributors. Clang — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-April-2019].

[100] Wikipedia contributors. Openjdk — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=OpenJDK&oldid=927329117, 2019. [Online; accessed 23-November-2019].

[101] Wikipedia contributors. Strip (unix) — Wikipedia, the free encyclopedia, 2019. [Online; accessed 24-April-2019].

[102] Wikipedia contributors. Dtrace — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DTrace&oldid=950798652, 2020. [Online; accessed 25-April-2020].

[103] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[104] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 219–229, New York, NY, USA, 1994. ACM.

[105] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 343–356, 2013.

[106] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[107] Adarsh Yoga and Santosh Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–501. ACM, 2019.

[108] Tingting Yu and Michael Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks.

In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400, 2016.

[109] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[110] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: generic off-cpu analysis to identify bottleneck waiting events. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 527–543, 2018.