StructSlim: A Lightweight Profiler to Guide Structure Splitting

Probir Roy Xu Liu

Department of Computer Science
College of William and Mary
Williamsburg, VA 23185 USA
{proy, xl10}@cs.wm.edu

Abstract

Memory access latency continues to be a dominant bottleneck in a large class of applications on modern architectures. To optimize memory performance, it is important to utilize the locality in the memory hierarchy. Structure splitting can significantly improve memory locality. However, pinpointing inefficient code and providing insightful guidance for structure splitting is challenging. Existing tools typically leverage heavyweight memory instrumentations, which hinders the applicability of these tools for real long-running programs. To address this issue, we develop StructSlim, a profiler to pinpoint top candidates that benefit from structure splitting.

StructSlim makes three unique contributions. First, it adopts *lightweight* address sampling to collect and analyze memory traces. Second, StructSlim employs a set of novel methods to determine *memory access patterns* to guide structure splitting. We also formally prove that our method has high accuracy even with sparse memory access samples. Third, StructSlim *scales* on multithreaded machines. StructSlim works on fully optimized, unmodified binary executables independently from their compiler and language, incurring around 7% runtime overhead. To evaluate StructSlim, we study seven sequential and parallel benchmarks. With the guidance of StructSlim, we are able to significantly improve all these benchmarks; the speedup is up to 1.37×.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes; D.2.8 [*Metrics*]: Performance measures

General Terms Measurement, Performance

CGO'16, March 12–18, 2016, Barcelona, Spain
ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854053

Keywords Data locality, address sampling, lightweight profiling, structure splitting

1. Introduction

Modern processors employ a hierarchy of caches to reduce the data access latency to main memory. Usually, caches physically located nearer to the CPU have higher data transfer bandwidth and lower access latency. To efficiently utilize the memory hierarchy, one needs to maintain good data locality to avoid frequent accesses to a far away memory. Memory access patterns that block data into the fastest caches and access it multiple times before it is evicted are said to have excellent data locality. There are two typical types of data locality: *temporal* and *spatial*. An access pattern exhibits temporal locality when it accesses the same memory words multiple times. An access pattern exploits spatial locality when it accesses a memory location and then accesses nearby locations soon afterward. Typically, spatial locality goes unexploited when accessing data with a large stride or indirection. While hardware prefetching can recognize non-unit accesses, long access strides cause low utilization of caches due to frequent capacity-based evictions.

Access patterns with poor spatial locality cause useless data to be loaded into cache. Such access patterns squander much of a processor's memory bandwidth loading values from main memory that will never be used before being evicted. Figure 1 shows an example. The original code creates an array of structures but uses only two fields in one loop and the other two fields in a second loop. As all the four structure fields; `a`, `b`, `c`, and `d` per array element reside in the same cache line, the computation in both loops only uses part of a cache line, wasting significant cache space and memory bandwidth. To address this performance issue, one can split structures [38] to optimize data layouts in memory – this approach is referred to as "structure splitting". If the fields of a structure that are frequently accessed together are known, one can reorganize such structure by splitting the structure into multiple structures each containing only the fields that have high affinity (i.e., frequently accessed together). For example, Figure 1 shows the code transformation after splitting structure `type` into `type1` and `type2`. With this optimiza-

```
struct type {int a; int b; int c; int d;};
struct type Arr[N];
for (i = 0; i < N; i++)
    B[i] = Arr[i].a + Arr[i].c;
...
for (i = 0; i < N; i++)
    C[i] = Arr[i].b + Arr[i].d;
```

The original code uses an array of structure Arr in a loop. The fileds `a` and `c` are accessed together in a loop, while `b` and `d` are accessed together in another loop.

```
struct type1 {int a; int c;};
struct type2 {int b; int d;};
struct type1 Arr1[N];
struct type2 Arr2[N];
for (i = 0; i < N; i++)
    B[i] = Arr1[i].a + Arr1[i].c;
...
for (i = 0; i < N; i++)
    C[i] = Arr2[i].b + Arr2[i].d;
```

The optimized code splits the structure into two for better spatial locality in both loops.

Figure 1: The code and data structures before and after structure splitting optimization.

tion, only useful data are loaded into caches in each loop, which efficiently uses cache space and bandwidth.

Identifying opportunities for structure splitting and making appropriate decisions for code transformation are difficult, especially for the programs that consist of hundreds of thousands of code lines and run with hundreds of threads. Developers need insightful guidance from compilers or performance tools to make decisions. There are principally two techniques to identify structure field affinity: static analysis and dynamic profiling. Static analysis [6, 24], which generates structure splitting decisions at compilation time, has three weaknesses. First, it cannot accurately handle pointers and aliases that are widely used for memory references. Second, it is difficult to provide optimization advice for the whole program which has multiple source files compiled separately. Third, it relies on specific compilers instead of working on the fully optimized binary code.

In contrast, dynamic profiling technique overcomes these shortcomings. However, many existing methods [8, 35, 38] incur high overhead. Typically, they instrument the code to analyze access patterns. For example, collecting the histogram of reuse distance [33] for each memory access to guide structure splitting can slow down the program by 153×. Computing affinities with access frequencies of each structure field shows lower overhead [35], but still more than 4×. To reduce the overhead, researchers have explored bursty sampling [37] to monitor a small subset of memory accesses as a representative of the whole program. Bursty sampling, however, still incurs 3-5× overhead [27], which hinders its applicability for long-run applications.

Processor models	Sampling techniques
Intel Nehalem	Precise event-based sampling with load latency (PEBS-LL)
Intel Itanium	Data event address register (DEAR)
Intel Pentium4	Precise event-based sampling (PEBS)
AMD Opteron	Instruction-based sampling (IBS)
IBM POWER5	Marked event sampling (MRK)

Table 1: The address sampling techniques in different processor models and their successors.

To address these issues in dynamic profiling, we develop StructSlim, a lightweight profiler that guides structure splitting. StructSlim makes the following three contributions:

- StructSlim adopts an efficient sampling-based data collection mechanism based on *hardware performance monitoring units*, which reduces the runtime overhead of StructSlim to ∼7% on average.
- StructSlim employs a set of novel methods to provide insightful guidance for structure splitting. Such methods include filtering out insignificant structures, accurately identifying field accesses, and computing latency-based field affinities.
- StructSlim works for both sequential and parallel applications, with scalable designs.

StructSlim works on fully optimized binary executables without manual instrumentation. StructSlim is independent from compilers and programming languages. To evaluate StructSlim, we study seven well-known benchmarks, covering both sequential and multithreaded code. Guided by StructSlim, we are able to identify structure fields with high affinities in the whole program. The optimization via structure splitting for these benchmarks yielded speedups as high as 1.37×.

We organize the remaining paper as follows. Section 2 introduces the background knowledge about the technique StructSlim uses for efficient memory profiling. Section 3 reviews the existing work on structure splitting. Section 4 describes our approaches in analyzing memory bottlenecks and providing structure splitting guidance. Section 5 shows the design and implementation of StructSlim that employs novel analysis techniques. Section 6 studies seven benchmarks to evaluate StructSlim. Section 7 presents some conclusions and previews some future work.

2. Background

To support lightweight analysis, StructSlim leverages address sampling that is available in most modern CPU architectures. In address sampling, performance monitoring units (PMUs) periodically select a memory access and monitor its execution through the pipeline. Table 1 lists the address sampling mechanisms in modern architectures. Typically, address sampling captures three pieces of information: (1) the instruction pointer (IP) of the sampled memory access, (2)

the memory address (effective address) the sampled instruction reads or writes, and (3) related memory events caused by the sampled instruction, such as cache or TLB misses. Among these sampling techniques, only PEBS-LL and IBS provide latency measure for the sampled memory access, which is necessary for StructSlim. Thus, StructSlim is built atop both Intel PEBS-LL and AMD IBS.

Advantages of address sampling: Compared to traditional measurement techniques, address sampling provides unique capabilities. Unlike instrumentation-based measurement [2, 27], address sampling incurs extremely low overhead, less than 3%. Because of the negligible distortion to the monitored program execution, the observed memory events (e.g., cache misses and latency) related to the sampled access can precisely quantify the execution bottlenecks. In contrast, to assess the cache misses and their latency, instrumentation-based methods require simulators, but are difficult to model complex memory architectures. Unlike event-based sampling (EBS) supported by traditional performance counters, address sampling records more information. EBS does not record the instruction pointer that triggers the event or the effective address. There is also no latency information captured. Thus, address sampling outperforms existing measurement techniques and provides potential opportunities for efficient and effective analysis of memory bottlenecks.

Challenges with address sampling: There are two challenges for the analysis based on the data collected by address sampling. First, address sampling only monitors a small subset of memory accesses. Thus, it requires some novel methods to extract access patterns hidden in the raw data to represent the profile of the whole program execution. Second, address sampling randomly monitors memory accesses, which has less flexibility than the instrumentation technique. With instrumentation, one can leverage the bursty sampling technique [37] to monitor all memory accesses in a code region or a time window, which facilitates access pattern analysis [22, 27]. However, address sampling does not support monitoring bursty memory accesses. Instead, the distance (in the number of memory accesses) between the two adjacent memory samples varies and is usually large. Thus, StructSlim cannot directly apply prior methods for access pattern analysis. In Section 4, we introduce a set of novel method to show their effectiveness and accuracy in access pattern analysis to guide structure splitting.

3. Related Work

There are numerous prior efforts focusing on data layout optimization in memory, but we only discuss the ones that are closely related to StructSlim. Section 3.1 reviews the prior approaches on structure splitting optimization; Section 3.2 reviews the lightweight memory profiling based on address sampling.

3.1 Structure Splitting Optimization

There are a number of compiler-based techniques [6, 10, 12, 16, 24, 26] to perform data layout optimization. They either rely on static analysis or depend on extra information generated by compilers, and they have three weaknesses compared to the dynamic analysis used by StructSlim. First, static analysis has limited capability in handling aliases and pointers. Second, compilers usually analyze the code at a granularity of files so it is difficult for them to give optimization advice based on the whole program. Third, their compiler-dependent features impede the usage.

To be independent from compilers, Chilimbi et al. [8] used field access frequency to compute field affinities. Zhong et al. [38] guide structure splitting by quantifying the affinity between structure fields via collecting reuse distance signatures of each structure field. Similar to StructSlim, their approach performs whole-program profiling that guarantees that the optimization can benefit the whole program execution rather than individual loops. However, computing the reuse distance incurs overhead [33] as high as 153×.

Yan et al. [35] developed ASLOP to identify structure splitting opportunities with the help of code instrumentation and hardware performance counters. It counts the execution frequency of basic blocks and cache misses to compute structure field affinities. ASLOP reduces overhead by saving instrumentation on every memory access. However, its overhead is still as high as 4.2×.

In contrast to these approaches, StructSlim does not instrument memory accesses or basic blocks, so it has much lower runtime overhead. Yan et al. [34] developed the most related work to StructSlim, which leverages the performance monitoring units to monitor the basic block execution with low overhead. Unlike StructSlim, their approach does not directly analyze memory addresses to quantify the affinities between fields of a data structure. Thus, their approach requires substantial manual efforts to identify structure splitting candidates. Moreover, their approach does not work for parallel programs.

Wang et al. [32] proposed on-the-fly structure splitting for heap data objects. They maintain a customized heap and leverage online data copying to perform structure splitting. However, their approach splits all the fields in a structure rather than computing their affinities, as known as maximal splitting [9, 36], which may lead to sub-optimal performance. Moreover, they do not perform whole program analysis to guide the structure splitting.

3.2 Lightweight Memory Profiling

Buck and Hollingsworth developed Cache Scope [4] to perform data-centric analysis using Itanium 2 EAR. Cache Scope associates latency with data objects and functions that access them. HPCToolkit [17, 19] enhances Cache Scope by attributing latency metrics to the full calling contexts of code and data. HPCToolkit collects data based on AMD IBS

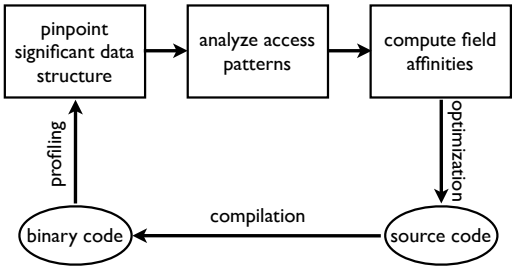

Figure 2: The workflow of StructSlim's analysis.

and IBM MRK. Memphis [23] and MemProf [14] leverages AMD IBS to associate latency with data structures to identify costly memory accesses to remote sockets.

Unlike StructSlim, these tools only collect and attribute address samples, they do not perform sophisticated, automatic pattern recognition of the sampled memory accesses. Some recent tools such as HPCToolkit-NUMA [18], ArrayTool [21], and ScaAnalyzer [20] perform analysis to guide data placement across sockets, array regrouping, and memory scaling losses, respectively. However, none of these tools can give advice for structure splitting. To the best of our knowledge, *StructSlim is the first lightweight profiling tool that provides insightful guidance for structure splitting in both sequential and parallel programs*.

4. StructSlim Methodology

In this section, we describe our approach which guides structure splitting with the data collected by address sampling. Figure 2 shows an overview of the analysis workflow which consists of three components. First, we filter out insignificant data structures from consideration to avoid wasting optimization efforts on inconsequential parts of the program. Second, we analyze access patterns to identify how different fields of a data structure are referenced in each loop. Finally, we aggregate access pattern analysis from all loops in the program and compute field affinities for the whole program. Programmers are required to do two tasks to achieve end-to-end optimization: (1) compile the source code with any compiler and any optimization option [1] and (2) transform the source code with the structure splitting guidance.

Before diving into the details of each analysis component, we first introduce the preprocessing stages with address samples, which are the foundation of other analyses. We preprocess address samples by associating them with code-centric and data-centric views during the program execution.

Code-centric attribution: Associating samples with code, such as instructions and loops, is important for StructSlim's analysis. It is straightforward to attribute samples to instructions, as the performance monitoring unit (PMU) captures the instruction pointer for each sample. In addition, StructSlim identifies loop boundaries via interval analysis [11] on the binary and attributes samples whose instruction pointers fall in the loop interval.

[1] `-g` is needed to map the analysis to the source code.

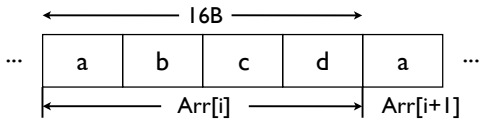

Figure 3: The stride of accessing a data structure field in a array of structure. This is an example for the first loop in Figure 1. Accessing field a with $Arr[i].a$ has a stride of the whole structure size, 16 bytes.

Data-centric attribution: StructSlim leverages existing techniques [19] to associate address samples with data structures. It records memory ranges allocated for data objects by reading symbols and overloading allocation functions. The names of static data objects in the symbol table and the allocation call paths for heap data objects are used to uniquely identify data objects. StructSlim does not monitor stack data objects, which usually cause trivial performance impact in the memory. Then, StructSlim leverages the effective address captured by PMU to attribute the sample to a data object. For data-centric attribution, StructSlim does not distinguish array of aggregate types or array of primitive types.

Along with the sample attributions, all events, such as cache misses and latency, associated with samples are aggregated to a specific line of code or a data structure. Based on these two sample attributions, StructSlim performs the three analyses in Figure 2. We describe each analysis in the following subsections.

4.1 Pinpointing Hot Data

StructSlim filters out data structures that have low access latency during the whole program execution, because optimizing these data structures yields little performance gain. With data-centric attribution, the memory access latency of each sampled memory access is associated with a data structure. Thus, StructSlim computes a metric l_d, as shown in Equation 1, for each data structure.

$$l_d = \frac{\sum_{data} l}{\sum_{program} l} \quad (1)$$

In this equation, l is the latency of a sampled access. $\sum_{data} l$ and $\sum_{program} l$ sum up latency associated with the investigated data structure and the whole program. l_d is between 0 and 1; a higher value means more significance of the data structure. From our experiments, we only need to investigate the top three data structures for optimization.

4.2 Analyzing Access Patterns

StructSlim performs access pattern analysis for each significant data structure in each individual loop. The goal of this analysis is to identify which fields of the examined data

structure are accessed in each loop. As discussed in Section 2, without monitoring every memory access, it is challenging to perform this analysis on binary with sparse samples: there is no easy way to know the structure size and there is no easy way to know which field of this structure is accessed by a specific memory instruction. StructSlim obtains this information by analyzing address samples based on one key observation.

> *Observation*: If a memory instruction keeps accessing one field of an aggregate data structure across loop iterations, this instruction shows a non-unit, constant stride access pattern. This stride is (a multiple of) the size of the structure.

Understanding this observation is straightforward. Taking the first loop in Figure 1 as an example, the memory access of $Arr[i].a$ has a stride of 16 bytes because the access needs to cross $Arr[i].b$, $Arr[i].c$, and $Arr[i].d$ to touch $Arr[i+1].a$ as shown in Figure 3. In addition, if loop induction variable i increments more than one per iteration, e.g, n, the stride for accessing field a is $16n$. From this observation, we can transform the analysis for the field access to a stride analysis. If we can compute the stride, we can assess the data structure size. Moreover, from the offset computation (shown later), we can know which field is accessed.

To perform the analysis, StructSlim makes the following assumption: an instruction in a specific calling context only accesses one field of a data structure. For a formal description, if an instruction i accesses an array starting at address A and this array object is of the structure type defined as fields f_1, f_2, ..., f_n, instruction i in one calling context, with this assumption, only accesses field f_i of this array of structure. In practice, this assumption holds most of the time on a large variety of code, especially scientific solvers. Based on this assumption, we identify streams (Section 4.2.1) and analyze their stride (Section 4.2.2).

4.2.1 Identifying Streams

We define *a stream* as a sequence of memory accesses to an array in a loop, corresponding to the loop induction variable. Based on the assumption, we simply say that any instruction in a loop that accesses a data structure is a stream. If the same instruction accesses multiple data structures, this instruction forms multiple streams. With the help of code- and data-centric attributions, we can identify all the streams in a loop and associate the instruction pointers as well as the involved data structures with them. A streams is a basic unit which we use for further analysis. Compared to the stream measurement by detailed memory instrumentation [22], our approach may incur some inaccuracy, but from our experiments, it can practically identify streams accurately with low overhead.

4.2.2 Analyzing Strides

Stride analysis computes the structure size and identifies the field that is accessed by each stream based on address samples. We develop a novel method — the GCD algorithm — in StructSlim for efficient stride analysis. We formally describe the GCD algorithm as follows:

Suppose m_1, m_2, ... m_k are k samples with unique addresses of a stream that references data object D in loop L. These k samples access different offsets of D. We use Equation 2 to compute the address difference d between each two adjacent sampled addresses.

$$d_i = |m_i - m_{i-1}| \quad (where, 1 < i \leq k) \tag{2}$$

Then we compute the greatest common divisor (GCD) of these address differences as the stride of accessing D in loop L using Equation 3.

$$stride = gcd(d_i) \quad (where, 1 < i \leq k) \tag{3}$$

Algorithm intuition: We take the array of structure Arr in Figure 3 for example. If a stream always accesses field a (e.g., $Arr[i].a$ in a loop with induction variable i), the two adjacent samples can access $Arr[m].a$ and $Arr[n].a$; the difference between the two sampled addresseses is obviously a multiple of the structure size, 16 bytes. With taking more samples (e.g., sampled $Arr[2].a$, $Arr[5].a$, and $Arr[7].a$), we expect the GCD of the address differences of these samples (e.g., 48 and 32 bytes) to be 16 bytes, reflecting the real stride of the access pattern.

Accuracy analysis: To formally quantify the accuracy of the GCD algorithm, we use $stride_r$ to denote real stride of the stream that accesses D in loop L. According to the definition of stride, $stride$ computed in Equation 3 is always a multiple of $stride_r$. For example, if only the addresses of $Arr[2].a$, $Arr[4].a$, and $Arr[6].a$ are sampled, $stride$ is computed as 32 rather than the $stride_r$ 16. To have $stride$ equal to $stride_r$ with high probability, there should be enough address samples for this stream. We then prove that actually, with a small number of samples per stream, the GCD algorithm can obtain $stride_r$ with a very high probability. To not lose generality, we quantify the accuracy for analyzing a loop with unit ($stride_r = 1$) stride. Equation 4 shows the accuracy of our stride analysis method with k samples of *unique* addresses out of total n addresses. $\binom{\frac{n}{i}}{k}$ is the number of ways all k samples leading to the computation of $stride$ as i, instead of 1.

$$\begin{aligned} accuracy &= 1 - \frac{\binom{\frac{n}{2}}{k} + \binom{\frac{n}{3}}{k} + \binom{\frac{n}{5}}{k} + ...}{\binom{n}{k}} \\ &< 1 - \frac{1}{2^k} - \frac{1}{3^k} - \frac{1}{5^k} - ... \end{aligned} \tag{4}$$

The equation subtracts all the possibilities of computing $stride$ that is not equal to 1. We only consider i of the

prime multiples of stride 1, because it also covers all the cases for composite multiples. From this equation, we can see that **if k is larger than 10, the accuracy can be higher than 99%**. For real stride $stride_r$ of different values, we can get a similar equation and conclusion. 10 unique address samples is a significantly small number for data structures that are frequently accessed. Hence, our approach ensures high accuracy.

It is worth noting that the GCD algorithm reports irregular access patterns with stride 1, not distinguishing from the unit stride. However, access patterns with stride 1, either regular or irregular, are not of interest for StructSlim because there is no structure splitting opportunity.

Computing structure size: As the type and size of a data structure does not change during the program execution, we aggregate all the streams that access the same data structure and calculate the structure size by taking GCD of their strides, as shown in Equation 5. In this equation, $stride_i$ is the stride computed for stream i in the whole program that is associated with this data structure. The accuracy analysis is similar to the aforementioned GCD algorithm.

$$size = gcd(stride_i) \tag{5}$$

Identifying accessed structure field: We use the offset of the field to the starting address of the data structure to uniquely identify it. For example, the offsets of fields a and b in the structure $type$ in Figure 1 are 0 and 4 bytes, respectively. We compute this offset for each stream with Equation 6. In this equation, m_i is the effective address of an arbitrary memory access in stream i; s is the starting address of the data structure in the memory, which is obtained from either monitoring data allocation functions or reading from symbol tables; $size$ is the size of the data structure obtained from Equation 5.

$$offset_i = (m_i - s) \ \ mod \ \ size \tag{6}$$

4.3 Computing Field Affinities

We compute the field affinities of data structures for the whole program execution. We compute the affinities between all fields in each significant data structure. Unlike previous approaches that count the number of memory accesses, we use the memory access latency to derive an affinity metric, which gives a more accurate memory access penalty to each structure field. We use two steps to compute the affinity metric.

First, we analyze each loop. We aggregate the latency incurred by all the streams in a loop that have the same offset in the same data structure, for the purpose of quantifying the accesses to a structure field in a loop. Second, we take all loops in the program into consideration and use Equation 7 to compute the affinity A_{ij} between field i and j of a structure.

$$A_{ij} = \frac{\sum lc_{ij}}{\sum l_{ij}} \tag{7}$$

In this equation, $\sum lc_{ij}$ is the aggregate latency of accessing field i and j in all the common loops that reference both fields, while $\sum l_{ij}$ is the total latency of accessing fields i and j in the whole program. Intuitively, when fields i and j are always accessed together in loops, $\sum lc_{ij}$ is a large proportion of $\sum l_{ij}$. A_{ij} is between 0 and 1; the larger A_{ij} is, the higher affinity is between stream i and j. We compute the affinity between each field pair. We cluster all fields with high affinities in a structure and generate the structure splitting guidance: regroup the structure fields with high affinities.

4.4 Handling Parallel Programs

To adapt the analysis for parallel programs with multiple threads or/and processes, we perform the aforementioned analysis for individual thread/process online and then merge the analysis results offline. StructSlim adopts a technique similar to HPCToolkit [19] to merge the code- and data-centric attributions in different threads. For code-centric attribution, we aggregate the metrics (e.g., latency) for the sampled memory accesses from different profiles with the same instruction pointer. For data-centric attribution, we aggregate the metrics with data structures of the same allocation site or the same name. A user may view the aggregate execution profile in a code- or data-centric manner, to focus either on hot code regions or hot data structures, respectively.

Based on the sample attributions, we perform all the analyses in Figure 2. For identifying significant data structures, we use the aggregate latency from different profiles to compute l_d for each data structure. For analyzing access patterns, we adapt Equation 5 to use access strides computed in different profiles to calculate the structure size. For computing field affinities, all latency metrics are computed with the aggregate latency from different profiles.

5. StructSlim Implementation

StructSlim consists of an online profiler and an offline analyzer. The profiler accepts a binary executable and monitors its execution. The analyzer processes the raw data collected during the profiling and associates the analysis with program structures and source code to provide guidance for structure splitting. The remaining section elaborates on the implementation of StructSlim's profiler and analyzer, and discusses the challenges we address for parallel programs.

5.1 Online Profiler

StructSlim uses `libmonitor` [13] to preload the profiling library into the address space of the binary executable. The libmonitor tool provides function callbacks before and after the program execution. In the program begin callback, the profiler sets up address sampling, while in the program end callback, the profiler reclaims all the resources allocated for the sampling and writes the profile into a file for further analysis. When an address sample is triggered, the profiler

Benchmarks	Suites	Application descriptions	Parallel
179.ART	SPEC CPU 2000 [29]	Neural Network based object recognision in a thermal image	No
462.libquantum	SPEC CPU 2006 [28]	Simulation of quantum computer	No
TSP	Olden [5]	Traveling Salesman Problem solver	No
Mser	The San Diego Vision Benchmark Suite [31]	Image analyser for face detection	No
CLOMP 1.2	Lawrence Livermore National Laboratory CORAL [15]	Designed to measure OpenMP and multi-threading performance issues	Yes
Health	The Barcelona OpenMP Task Suite [1]	Columbian health care simulation	Yes
NN	Rodinia 3.0 [7]	Find k-nearest neighbour from unstructured data set	Yes

Table 2: Benchmark descriptions.

Benchmarks	Original execution time	Execution time after structure splitting	Speedups	StructSlim's measurement overhead
179.ART	17.1s	12.5s	1.37×	2.05%
462.libquantum	9.6s	8.8s	1.09×	2.79%
TSP	38.3s	35.1s	1.09×	2.42%
Mser	28.6s	27.7s	1.03×	2.95%
CLOMP 1.2	20.8s	16.6s	1.25×	16.1%
Health	49.7s	44.2s	1.12×	18.3%
NN	11.9s	8.9s	1.33×	5.21%
average	-	-	1.18×	7.1%

Table 3: Execution speedups after structure splitting guided by StructSlim and the measurement overhead incurred by StructSlim. For each data, we report its average value from three executions.

Benchmarks	L1 miss reduction	L2 miss reduction	L3 Miss reduction
179.ART	46.5%	51.1%	5.5%
462.libquantum	49%	82.6%	-637.9%
TSP	13.3%	19.9%	30.7%
Mser	8.3%	8.4%	36.7%
CLOMP 1.2	15.5%	26.4%	-2.3%
Health	66.7%	90.8%	-35.8%
NN	87.2%	98.0%	9.3%

Table 4: Cache miss reduction after optimizing benchmarks with structure splitting.

uses an interrupt handler to capture it. The handler records the instruction pointer, effective address, and the data latency associated with each sample.

With these raw data, the profiler performs online code- and data-centric attribution. With the help of `hpcstruct`, a tool from HPCToolkit, the profiler identifies loops and attributes samples. To attribute samples to data structures, the profiler uses libmonitor to overload memory allocation functions and uses symtabAPI [3] to read symbol tables from the binary. The profiler performs the GCD algorithm online to compute the stride for each stream.

Scalability: To scale the data collection and online analysis of the profiler, we design the profiler to monitor each thread individually, without any synchronization. Thus, StructSlim efficiently attributes samples in parallel programs to code and data structures. StructSlim analyzes access patterns for each thread separately and writes the analysis result to a profile file per thread.

5.2 Offline Analyzer

StructSlim's analyzer accepts the profiles generated by the profiler. It computes field affinities based on the access patterns collected by the profiler. The analyzer extracts the binary-to-source code line mapping information from debugging sections recorded in the binary and maps the analysis results, such as the allocation of data structures and significant loops, to the source code. Finally, StructSlim's analyzer outputs this high-level analysis as advice for structure splitting. The structure splitting advice is generated as a dot graph: the nodes are offsets of structure fields and the undirected, weighted edges show the field affinities. StructSlim clusters all the nodes into subgraphs; all the edges in a subgraph have high weights; and each subgraph is a new structure after the splitting optimization.

StructSlim's offline analyzer needs to merge all of the profiles from different threads and processes. If the number of threads and processes is huge, merging their profiles can be time consuming. To expedite this process, StructSlim leverages the reduction tree algorithm [30] to merge all profiles in parallel.

6. Experimental Evaluation

We evaluate StructSlim on an Intel Xeon machine that has 16 2.6GHz Xeon E5-4650L processors. The machine has a private 32KB L1 data cache and a private 256KB L2 cache for each core, as well as a 20MB shared L3 cache. We study

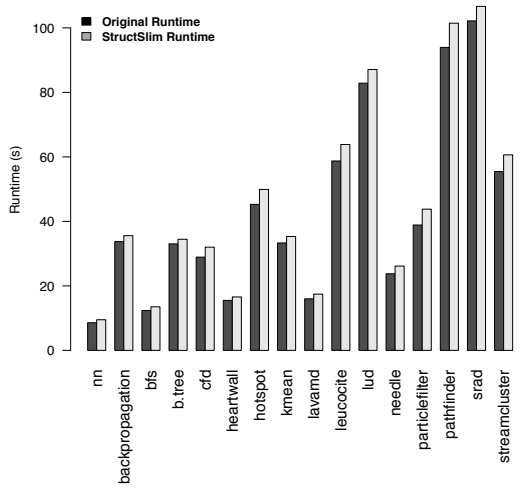

Figure 4: The runtime overhead of StructSlim when monitoring Rodinia benchmarks.

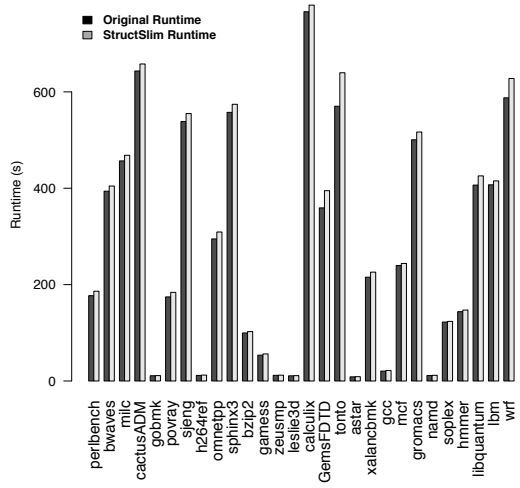

Figure 5: The runtime overhead of StructSlim when monitoring SPEC CPU 2006 benchmarks.

seven sequential and parallel benchmarks from a few well-known suites, as shown in Table 2. All these benchmarks are compiled with `gcc 4.8 -g -O2`. StructSlim uses Intel PEBS-LL PMUs to take an address sample for every 10,000 memory accesses. We run all parallel benchmarks with four threads in the same socket to avoid performance impact from multiple memory controllers and socket interconnects.

StructSlim overhead We use StructSlim to monitor all Rodinia and SPEC CPU 2006 benchmarks and report the runtime overhead in Figure 4 and 5. Our experimental results show that StructSlim incurs only ∼8.2% and ∼4.2% overhead on average for Rodinia and SPEC CPU 2006 benchmarks respectively.

Optimization overview Table 3 provides an overview of the speedups for each benchmark from the structure splitting optimization guided by StructSlim. All benchmarks receive significantly improvement in their end-to-end execution time, as high as 1.18× on average. StructSlim incurs only ∼7% runtime overhead to collect the performance data for the optimization guidance of the reported benchmark applications.

Field	I	W	X	V	U	P	Q	R
Latency	5.5%	2%	3.7%	3.7%	7.1%	73.3%	4.7%	0%†

† 0% means that StructSlim does not capture the access to field R via address sampling.

Table 5: StructSlim's access pattern analysis of ART to identify fields of `f1_neuron` Structure. StructSlim also associates the access latency to each field in the percentage of the total latency accessing this structure.

To verify that most of the performance gains are from the improvement of data locality, we collect cache miss events using hardware performance counters. Table 4 shows that cache misses are substantially reduced in each cache layer, except the L3 misses for libquantum and Health. We investigated them and found that most of their memory accesses are served in caches; few accesses go beyond caches to main memory. Thus, the rise in L3 misses can be deemed as system noise and has negligible impact to the program performance. In the remaining section, we explain the utility of StructSlim on each benchmark.

6.1 SPEC CPU 2000 - ART

With the help of the data-centric attribution and the derived latency metric l_d, StructSlim identifies that an array of structure, *f1_neuron*, used in ART accounts for 80.4% of total memory access latency. StructSlim further decomposes the latency to different fields of *f1_neuron* when analyzing the access patterns, as shown in Table 5. Each field, except `R`, accounts for significant latency; among them, field `P` is the hottest, with 73.3% of total latency.

We show StructSlim's code-centric analysis in Table 6. It associates latency with each significant loops and identifies the fields accessed in each loop. From this table, we can see which fields are always accessed together in hot loops. For example, the hottest loop of line 615-616 accounts for 56.8% of total latency and only field *P* is accessed in this loop. Another hot loop in line 559-570, both field `X` and `Q` are accessed.

StructSlim quantifies field affinities in Figure 6. We see that fields *I* and *U* have a high affinity, 0.86, so StructSlim recommends to group these two fields in a new structure. The fields `X` and `Q` also show a high affinity, which also suggests to group them in a new structure. On the other hand, although *P* and *U* are accessed together in two loops (as shown in Table 6), they have a low affinity 0.05, because these two loops account for a smaller amount of latency compared to the loops that only access `P`.

Loops with line numbers	Latency percentage	Accessed fields
131-138	1.59%	U,P
559-570	8.42%	X,Q
553-554	1.98%	W
545-548	10.83%	U, I
615-616	56.57%	P
607-608	14.40%	P
589-592	2.25%	U, P
575-576	3.72%	V
1015-1016	0.24%	I

Table 6: StructSlim associates latency with structure fields of `f1_neuron` in each monitored loop of ART.

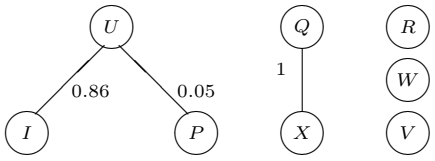

Figure 6: StructSlim generates the affinity graph for the fields in structure `f1_neuron`.

```
typedef struct {double P;} f1_neuron_P;
typedef struct {double Q; double X;} f1_neuron_XQ;
typedef struct {double U; double *I;} f1_neuron_IU;
typedef struct {double V;} f1_neuron_V;
typedef struct {double W;} f1_neuron_W;
typedef struct {double R;} f1_neuron_R;
```

Figure 7: Structure splitting of `f1_neuron`.

According to the analysis of StructSlim, we split the structure of `f1_neuron` to six new structures, as shown in Figure 7. This optimization yields a 1.37× speedup.

6.2 SPEC CPU 2006 - libquantum

For libquantum, StructSlim identifies an array of the type *quantum_reg_node_struct* accounts for 99.9% of total memory access latency. *quantum_reg_node_struct* has two fields in this structure: *amplitude* and *state*; accessing field *state* accounts for ∼100% of the access latency of this structure. StructSlim's code-centric analysis pinpoints three hot loops at line 170-174, 89-98 and 61-66 that access *state* and respectively account for 43.4%, 40.8% and 15.5% of the total latency incurred by *quantum_reg_node_struct*. As these hot loops do not access *amplitude*, StructSlim reports the affinity between these two fields as 0. According to StructSlim's analysis, we split structure *quantum_reg_node_struct* as shown in Figure 8, which leads to a 1.09× speedup.

6.3 Olden - TSP

StructSlim identifies that arrays with the *tree* structure account for 100% of the total memory access latency. The structure has seven fields: `sz`, `x`, `y`, `left`, `right`, `next`, `prev`. StructSlim further shows that fields *x*, *y* and *next* account for 14.4%, 4.9% and 80.7% of the entire access latency to this data structure. In addition, StructSlim pinpoints

```
struct quantum_reg_node_struct_a {
  COMPLEX_FLOAT amplitude;};
struct quantum_reg_node_struct_s {
  MAX_UNSIGNED state;};
struct quantum_reg_node_struct {
  struct quantum_reg_node_struct_a * struct_a_ptr;
  struct quantum_reg_node_struct_s * struct_s_ptr;};
```

Figure 8: Structure splitting of `quantum_reg_node_struct` in libquantum.

```
struct tree_0 { double x,y; int next;};
struct tree_1 { int sz; int left, right; int prev;};
```

Figure 9: The optimized structure of `tree` after structure splitting in TSP Benchmark.

```
idx_t *GNode_parent_pt;
typedef struct {idx_t shortcut; idx_t region;
 int area;} node_t;
```

Figure 10: Structure splitting for `node_t` in MSER.

that these three fields are accessed together in the loops at line139-142 and 170-173, associated with 23.4% and 76.6% of the total access latency, respectively. StructSlim quantifies that these three fields have high affinities of 1 and gives recommendations to group them into one new structure. Thus, we split these fields from the rest of the four fields in *tree* as shown in Figure 9 and achieve a speedup of 1.09× for the whole program.

6.4 SD-VBS - MSER

StructSlim identifies three arrays used in MSER but only the array of *node_t* is significant, accounting for 21.2% of total memory access latency incurred during the program execution. StructSlim's stride analysis pinpoints field *parent* alone (offset 0 with stride 16) is frequently accessed in a hot loop at line 679-683. Thus, we split the structure to separate `parent` as shown in Figure 10 and obtain a 1.03× speedup of the whole program.

6.5 CORAL - CLOMP

CLOMP is designed to measure OpenMP overhead and performance impact due to parallel threads. We run CLOMP with four OpenMP threads. StructSlim's latency metric l_d shows that the array of the structure type *_Zone* incurs 89.1% of the total memory latency throughout the benchmark execution. This array is allocated by one thread but accessed by all of threads in parallel. StructSlim merges the accesses to this array from different threads. With further analysis, StructSlim shows that the loop at line 328-337 accounts for all access of this array. Stride analysis suggests that only fields *value* and *nextZone* are accessed in this loop. Field

```
struct _ZoneHeader {long zoneId; long partId;};
struct _Zone {double value; struct _Zone *nextZone;
              struct _ZoneHeader *head; };
```

Figure 11: The optimized structure of _Zone after structure splitting in CLOMP Benchmark.

```
struct Patient_super_struct {
  struct Patient_super_struct *forward;
         struct Patient *this; };
struct Patient {
  int id; int32_t seed; int time; int time_left;
  int hosps_visited; struct Village *home_village;
  struct Patient_super_struct *back;};
```

Figure 12: Structure splitting for `Patient` in Health.

value accounts for 44.7% of the total access latency of the loop, while *nextZone* accounts for 55.3%. StructSlim computes the affinity between these two fields as 1 but 0 affinity with the other two fields `zoneId` and `partId`. StructSlim recommend to split this structure and group `value` and `nextZone` together. Figure 11 illustrates the data structure after splitting. We create a new field *head* to keep track of the two fields *zoneId* and *partId* that are split to a new structure `_ZoneHeader`. As a result we achieve a 1.25× speedup for the whole CLOMP Benchmark.

6.6 BOTS - Health

Health is implemented with OpenMP tasking constructs. We run Health with four threads. StructSlim identifies two arrays of structure but its derived latency metric shows that only *Patient* is significant, accounting for 95.2% of total memory access latency. Structure *Patient* has eight fields. Further access pattern analysis by StructSlim shows that only field *forward* is accessed in a hot loop at line 96. The affinity analysis reports that *forward* has low affinities with other fields so it recommends spliting field *forward* from other fields. Figure 12 shows the resulting structure splitting. This suggested optimization leads to a speedup of 1.12×.

6.7 Rodinia - NN

NN is an OpenMP program. StructSlim monitors eight arrays but only one array of the structure *neighbor* accounts for ∼100% of total access latency. Structure *neighbor* has two fields: *entry* and *dist*. StructSlim reports that fields *dist* and `entry` respectively account for 99.1% and 0.9% of the total access latency to this structure. Further analysis by StructSlim reveals that the hot loop at line 117-120 references *dist* but not `entry`. The affinity between the two fields is 0 so StructSlim recommends to split the two fields for better performance. Figure 13 shows the result of structure splitting. We optimize NN with the guidance of StructSlim and obtain a 1.33× speedup for the whole program.

```
struct neighbor_entry { char entry[REC_LENGTH];};
struct neighbor_dist { double dist;};
```

Figure 13: Structure splitting for `neighbor` in NN.

7. Conclusions and Future Work

In this paper, we described the design and implementation of a lightweight memory analysis tool, StructSlim, which identifies an important data layout optimization opportunity—structure splitting—in both sequential and parallel programs. To the best of our knowledge, StructSlim is the first to collect address samples via efficient modern performance monitoring units to perform structure splitting analysis. StructSlim adopts a set of novel methods to analyze address samples and address challenges, such as accurate access pattern analysis based on sparse, random address samples. To evaluate StructSlim, we applied it on seven sequential and parallel benchmarks from well-known suites. The experimental results show that StructSlim incurs around 7% of runtime overhead, on average. Furthermore, StructSlim provides insightful guidance to optimize these benchmarks with structure splitting and yields a 1.18× speedup on average. In our experience, it is easy to use StructSlim, and the feedback provided by StructSlim both visually and quantitatively guide optimization of code performance. The manual effort in restructuring the code was minimal. StructSlim's output can be easily consumed by a compiler pass such as ROSE [25] to perform profile-guided data-layout optimization.

As our future work, we will extend StructSlim to identify data layout optimization opportunities beyond structure splitting, such as array regrouping and data reorganization.

Acknowledgements

We thank Milind Chabbi, Ronak Buch, and Ed Novak for helping on the paper. This research was supported by the National Science Foundation (NSF) under Grant No. 1464157.

References

[1] The Barcelona OpenMP Task Suite. `https://pm.bsc.es/projects/bots`.

[2] K. Beyls and E. H. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62 –71, Feb. 2009.

[3] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the Intel ltanium 2 processor. In *SC*, 2004.

[5] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-memory Machines*. PhD thesis, Princeton, NJ, USA, 1996.

[6] G. Chakrabarti, F. Chow, and L. PathScale. Structure layout optimizations in the open64 compiler: Design, implementation and measurements. In *Open64 Workshop at the Inter-*

national Symposium on Code Generation and Optimization, 2008.

[7] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009.*, pages 44–54, Oct 2009.

[8] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, pages 13–24. ACM, 1999.

[9] S. Curial et al. Mpads: Memory-pooling-assisted data splitting. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 101–110, New York, NY, USA, 2008. ACM.

[10] M. Hagog and C. Tice. Cache aware data layout reorganization optimization in gcc. In *Proceedings of the GCC Developers Summit*, pages 69–92, 2005.

[11] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997. .

[12] R. Hundt et al. Practical structure layout optimization and advice. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 233–244, 2006.

[13] M. W. Krentel. Libmonitor: A tool for first-party monitoring. *Parallel Comput.*, 39(3):114–119, Mar. 2013.

[14] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A memory profiler for NUMA multicore systems. In *USENIX ATC*, 2012.

[15] Lawrence Livermore National Laboratory. LLNL Coral Benchmarks. `https://asc.llnl.gov/CORAL-benchmarks/`.

[16] J. Lin and P.-C. Yew. A compiler framework for general memory layout optimizations targeting structures. In *Proceedings of the 2010 Workshop on Interaction Between Compilers and Computer Architecture*, INTERACT-14, pages 5:1–5:8, New York, NY, USA, 2010. ACM.

[17] X. Liu and J. Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proc. of the 9th IEEE/ACM Intl. Symp. on Code Generation and Optimization*, pages 171–180, Washington, DC, 2011.

[18] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *PPoPP*, 2014.

[19] X. Liu and J. M. Mellor-Crummey. A data-centric profiler for parallel programs. In *SC*, 2013.

[20] X. Liu and B. Wu. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *SC*, 2015.

[21] X. Liu, K. Sharma, and J. Mellor-Crummey. ArrayTool: a lightweight profiler to guide array regrouping. In *PACT*, 2014.

[22] G. Marin, C. McCurdy, and J. S. Vetter. Diagnosis and optimization of application prefetching performance. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 303–312, New York, NY, USA, 2013. ACM.

[23] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. of 2010 IEEE Intl. Symp. on Performance Analysis of Systems Software*, pages 87–96, Mar. 2010.

[24] V. T. Prashantha NR and V. N. Implementing data layout optimizations in the LLVM framework, 2014. URL `http://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf`.

[25] D. QUINLAN. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[26] E. Raman et al. Structure layout optimization for multithreaded programs. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 271–282. IEEE Computer Society, 2007.

[27] A. Rane and J. Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *PACT*, 2012.

[28] SPEC Corporation. SPEC CPU2006 benchmark suite. `http://www.spec.org/cpu2006`. 3 November 2007.

[29] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite. `http://www.spec.org/cpu2000/`. 29 April 2005.

[30] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *SC*, 2010.

[31] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. Sd-vbs: The san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society.

[32] Z. Wang, C. Wu, P.-C. Yew, J. Li, and D. Xu. On-the-fly structure splitting for heap objects. *ACM Trans. Archit. Code Optim.*, 8(4):26:1–26:20, Jan. 2012.

[33] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: A higher order theory of locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 343–356, New York, NY, USA, 2013. ACM.

[34] J. Yan, W. Chen, and W. Zheng. PMU guided structure data-layout optimization. *Tsinghua Science and Technology*, 16(2): 145–150, 2011.

[35] J. Yan, J. He, W. Chen, P.-C. Yew, and W. Zheng. ASLOP: A field-access affinity-based structure data layout optimizer. *Science China Information Sciences*, 54(9):1769–1783, 2011.

[36] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. Forma: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.

[37] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 91–100, New York, NY, USA, 2008. ACM.

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