AutoCPA: Automatic Continuous Profiling and Analysis

by

Zahra Rezapour Siahgourabi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Abstract

Poor data locality is a performance bottleneck in modern applications. The hierarchy of caches exiting in computer processors reduces data access latency from the main memory. However, inefficient cache utilization results in data cache miss overhead. Applications usually make frequent accesses to far away data that neglects the locality in the memory hierarchy. One approach to boost applications' performance is to reorder structure fields in a manner that efficiently utilizes the cache. To do so, extensive program-wide information is needed to gain insight about the access frequencies and access patterns of data.

This thesis introduces AutoCPA, which exploits hardware performance monitoring counters to find optimization opportunities in target applications, and provides insightful guidance for structure reordering. This system is a low-overhead and easy-to-use toolchain that uses a sampling-based approach to collect and analyze memory traces. Moreover, it generates a prioritized set of reordering that can improve cache utilization and locality. The recommendations for the optimal structure layout provided by this tool are obtained from multiple cache analysis algorithms implemented in AutoCPA. Performance results obtained by running AutoCPA on two widely-used applications, Redis and Memcached, illustrate the benefit of the implementation. These results confirm the general performance improvement of applications, with up to 10% instruction per cycle increase in Redis operations and 7.1% cache miss reduction in Memcached.

Acknowledgements

My sincere gratitude goes to my supervisor Dr. Ali Mashtizadeh for his guidance, critique, and insights to make this work successful. I learned a lot from him and it was a privilege to work under his supervision.

Thanks are also owed to the members of the compiler group of Huawei Technologies Canada, who supported this research with their insightful suggestions that broadened my perspectives.

I would also like to thank the members of my dissertation committee, Dr. Samer Al-Kiswany and Dr. Omid Abari, for taking the time to read my thesis and provide me with their encouraging comments and feedback.

Finally, I thank people who mean a lot to me, my parents and my brothers, for showing faith in me and giving me the liberty to choose what I desired. I am forever indebted to you all for giving me the opportunities and experiences that have made me who I am.

Dedication

This thesis is dedicated to my very special person, my best friend, Soroush who has been supporting and encouraging me during my pursuit of Master degree. I consider myself the luckiest in the world to have such a lovely and caring husband, standing beside me with your unconditional support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data locality plays an important role in the performance of many of today's popular applications. Excessive cache miss rates occur when a program accesses a large data set or touches certain data in a manner that fails to make use of locality. The performance bottleneck is exacerbated by the growing gap between processor and memory speed, causing the CPU to stay idle for long periods of time.

One approach to optimizing data cache utilization and improving data locality is to rearrange the layout of structures through field reordering. In this approach, extensive program-wide information about the access frequency and access pattern of structure fields is used to determine a new order of fields that makes the application more cache-friendly. However, collecting such information and identifying optimization opportunities manually increases in cost and inefficiency proportional to any increases in the size of a project's codebase. One solution is to use online profiling and collect profile data during the runtime. This data can then be analyzed to provide insights about the target application. Thus, the idea of having a background daemon to continuously monitor and report cache usage problems automatically is promising.

Continuous profiling of production workloads was first implemented in the DEC continuous profiler [2] (DCPI) on the Alpha processor. The use of hardware customizations made it possible to collect the profiles of production machines with low overhead. Until recently, profiling on all other architectures has been costly enough to prevent the development of a similar tool. The Google wide profiling [16] is a modern system used to implement continuous profiling through random sampling. This more recent system does not work on resource-constrained devices since it requires tremendous resources to keep all the profiling information.

New performance monitoring counters (PMC) hardware lets us revisit the DCPI work in modern architectures. These architectures, such as Intel x86, ARM ARMv8, and IBM POWER, present dozens of performance counters and architectural features that are incomprehensible to most programmers. These modern processors are complex, and programmers struggle to interpret the relevance of each performance metric and to utilize the PMC to improve their applications.

Through the careful analysis of performance monitoring features present in modern architectures, it is possible to build a profiler that can receive reports from the CPU and build a suitable data structure to track and store raw samples of problematic code segments continuously. By analyzing all the data collected from programs running in production machines, abundant insights can be gained into performance degradation, including from branch mispredictions and cache misses. That information can then be processed to explore optimization opportunities, which can then be presented to developers in a meaningful way. The value of such a system is even more considerable when the performance of large highly-optimized applications is further improved using the insights gained by the profiler's analysis.

In other words, we would like to have a system that accomplishes three objectives: (I) it continuously collects profile data with low overhead; (II) it represents the data in a compact form to suite resource-constrained devices, e.g., mobile phones; (III) it provides performance insights into systems automatically. These capabilities are important since a resource-oriented tool can be extended for use on any computing device, from mobile phones to the devices in production environments. Additionally, since the codebase of software changes frequently in the development life cycle, having an automated optimization tool helps to continuously update the code and solve optimization problems without the need for human interference.

This thesis introduces AutoCPA, a fully-automated low-overhead profiler that recommends structure reordering. This novel system, which uses the new features offered in modern architectures, makes continuous profiling practical. Taking sampling performance counters enables the collection of higher fidelity profiling information than was previously possible. AutoCPA works on highly optimized executables and is independent from compilers. It uses multiple cache analysis algorithms to find the optimal structure layouts while considering the whole target program. It then generates a prioritized set of actionable improvements for developers. AutoCPA has been evaluated on two well-known applications, Redis and Memcached, and was able to identify the access frequency of their structures. Guided by optimization techniques, AutoCPA was able to recommend structure reorderings and achieved up to 10% Instruction Per Cycle (IPC) improvement in Redis commands.

This research makes the following contributions:

- It uses modern hardware performance counters to build a low-overhead sampling-based system that collects data continuously from a wide variety of computing devices, from mobile phones to production machines.
- It has automated the optimization process to generate a prioritized set of actionable cache utilization improvements for developers through novel profile analysis algorithms.
- It has implemented and evaluated the new system on popular real-world applications, achieving up to 10% IPC improvement in Redis operations and 7.1% cache miss reduction in Memcached.

The rest of this thesis is structured as follows: Chapter 2 presents an overview of relevant works in the area of continuous profiling and structure layout optimization. Chapter

3 presents the design of the sample collection tool and the profile analysis and recommendation tool. Implementation details, such as details about hardware and software, are presented in Chapter 4. Chapter 5 introduces the evaluation steps and the performance metrics for structure reordering recommendations. Then, it shows the results of running AutoCPA with two large applications. Finally, Chapter 6, concludes the thesis by summarizing the key advantages of AutoCPA and suggesting some of the potential development opportunities.

Chapter 2

Related Work

Related work can be categorized into two main topics: continuous profiling and structure layout optimization.

2.1 Continuous Profiling

Online profiling is done through several methods. Some methods take advantage of performance monitoring unit (PMU) of CPU and a sampling-based approach to gather system-wide profiles. For example, continuous profiling of production workloads was first implemented in the DEC continuous profiler [2] (DCPI) on the Alpha processor. Authors of this work utilized hardware customizations that made it possible to collect profiling of production machines with low overhead. However, profiling on all other architectures has been extremely costly, and thus no similar tool has developed for other architectures until recently. Unlike the DEC Alpha which has fairly simple performance metrics, AutoCPA leverages the complex performance counters of modern architectures.

Google wide profiling [16] is a modern system to implement continuous profiling through random sampling, and it was built based on the assumption of having massive resources to keep all the profiling information. In contrast, AutoCPA uses an efficient data representation and storage which makes it suitable to be used on resource constrained devices, e.g., mobile phones.

Specialized hardware support is also proposed in some works to improve profiling using PMU. For example, ProfileMe [6] introduces specialized hardware support for Alpha processors that makes instruction sampling more accurate. Likewise, a novel hardware support is presented in [12] to identify and monitor program hotspots. However, their hardware does not exist in today's popular processors.

In contrast, some works have been proposed that do not use the hardware PMU. In [22], statistical sampling of the program counter is collected on clock interrupts. On the other hand, the work proposed by Conte et al. [5] takes samples from the contents of the branch-prediction hardware by using kernel-mode instructions. Unfortunately, these methods are not effective in providing fine-tune optimizations, since they do not collect accurate instruction frequency profiles.

Some methods propose online profiling by instrumenting programs in different ways. For example, Ammons et al. use a method which instruments programs to read performance counters. In this method, context-sensitive profiles are collected from a running application[1]. Alternatively, others have proposed dynamically modifying the program binary to capture branch executions [19]. However, the instrumentation used by these methods is not applicable in a production environment.

2.2 Structure Layout Optimization

There are numerous prior works that focus on data type layout optimization in memory. However, we only discuss the most related efforts to AutoCPA.

Some techniques provide data layout optimizations by relying on compiler-based information, or using static analysis [3, 8, 9, 11, 14, 15]. However, they have two major drawbacks comparing to the dynamic analysis methods used by AutoCPA: (I) static analysis has limitations in handling aliases and pointers; (II) compilers usually do file-level analysis, and it is very difficult for them to do optimizations considering the whole program.

Apart from compiler-based efforts, Chilimbi et al. [4] used the same approach as AutoCPA in taking advantages of field access frequency to generate optimization improvements based on field dependencies. Zhong et al. [23] performed structure splitting by quantifying field affinities through computing reuse distance signatures of structure fields. Although these approaches consider the whole program and recommend the global optimization improvements, the computation of reuse distance adds considerable overhead to the system [21].

Another related work to AutoCPA is StructSlim [13] that is a light-weight profiler for binary code, built on top of HPCToolkit. However, unlike AutoCPA, it optimizes memory latencies by structure splitting through identifying memory accesses.

Chapter 3

Design

3.1 System Overview

AutoCPA is a fully automated profiling tool that offers source code level optimizations to developers. It consists of two main components: a low-overhead sampling-based continuous profiler and a profile analysis tool that further processes the gathered profiles offline. Figure 3.1 gives an overview of the system.

AutoCPA relies on two key facts. First, modern architectures offer performance monitoring features that can track poorly performing parts of software programs. Secondly, there are always some optimization opportunities that either cannot be done by compilers or are simply overlooked by developers, especially as a project's codebase gets bigger. Thus, AutoCPA works as a background daemon to automatically monitor and report such problems in a meaningful way to developers.

AutoCPA can easily be run in the OS kernel and user space of any system and record useful performance-related data of all the running executables. It is also scalable from mobile phones to desktop computers and servers. AutoCPA takes advantage of BSD profiling infrastructure to sample from hardware performance counters. These samples and their

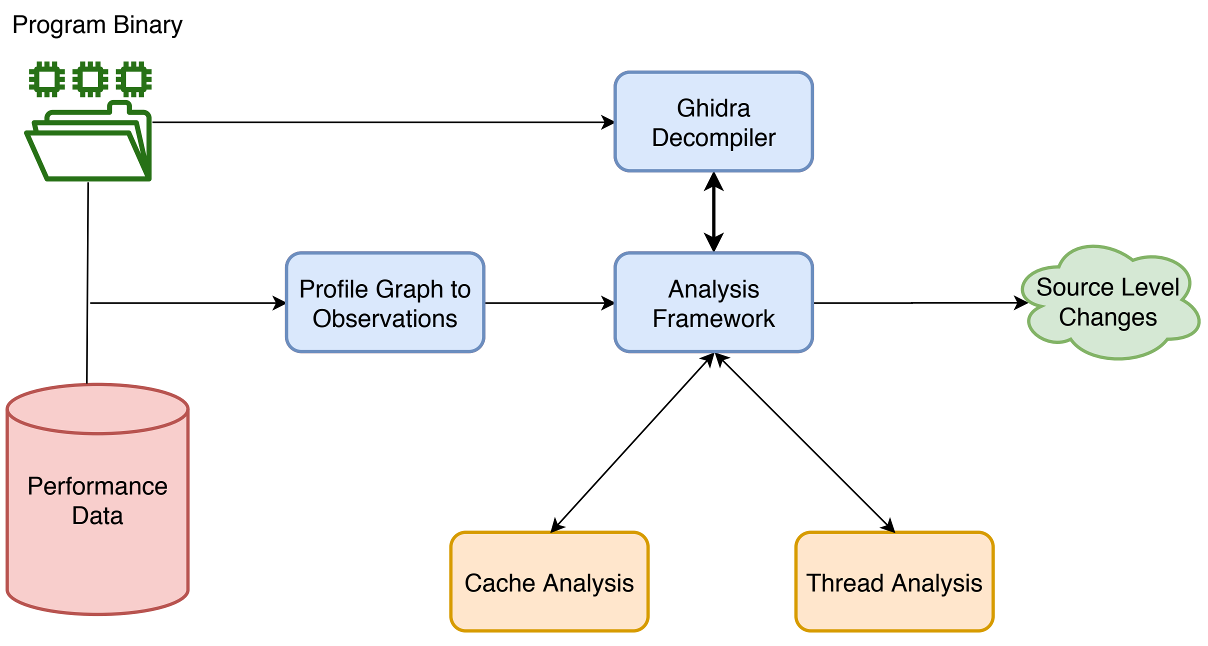

Figure 3.1: Overview of the AutoCPA system.

metadata are then periodically stored in the form of compact output files. The profile analysis tool, built on top of Ghidra framework[7], then takes all the raw samples related to the specified time period, and attributes them to actual source code symbols and structures. Finally, using its analysis algorithm, cache analysis and multi-threaded analysis, AutoCPA converts the symbolized samples to useful human-readable suggested optimizations in the source code. The resulting output of AutoCPA also represents the amount of benefit one can gain by applying the changes suggested by the system.

The next few sections present the components that contribute to the design of AutoCPA in detail.

3.2 Low Overhead Continuous Profiling

Our profiling tool consists of two main components: a data collection agent running as a daemon that communicates with the CPU to receive periodic samples, and a post-processing command-line tool that queries the profile records stored previously by the agent.

The profiling infrastructure is easy to use, and no specific setup is needed for the target system. The operating system kernel, runtime libraries, and user programs can be used as-is without the need to link against additional libraries or to recompile. The overhead of this toolchain is low. Even when the system is at full utilization, a reasonable amount of information can still be collected by the daemon agent, generally using less than five percent of the CPU.

3.2.1 Collecting Samples from Hardware PMC

AutoCPA's analysis mechanism relies on profiles that are collected as the workload executes. The profiling data is collected by periodically sampling the performance counters. This data contains the precise value of the program counter, its corresponding executable, its thread number, and the call stack of the program at capture time. One of the key features of our continuous profiling tool is its efficiency in terms of CPU overhead and space: it takes less than five percent CPU usage even when the system is at full utilization. The profile data is saved in compact form on disk, taking the least amount of space.

Gathering profiles relies on sampling from hardware performance counters available by modern CPUs. There are various types of performance counters, and depending on the hardware, each of them counts different events. Modern CPUs can be set up to trigger an interrupt when one or more performance counters reach a specific value. Issuing this interrupt means that after a certain number of events occur, a sample is taken and all its relevant information is captured. Gradually, if the sampling rate is high enough, and sampling is done over a long period, the poorly performing code segments appear

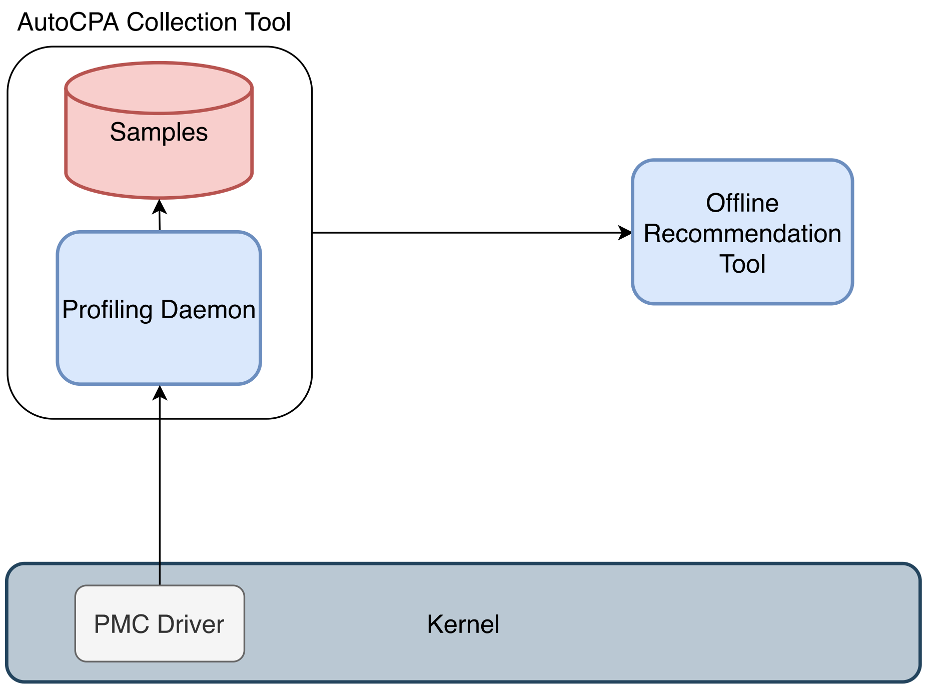

Figure 3.2: AutoCPA's collection tool continuously receives performance data from interacting with the Performance Monitoring Counter (PMC) module of the Kernel, which manages the in-CPU performance counters. This data is stored in suitable data structures and is then passed to the offline analysis and recommendation tool for further process.

in statistics. These statistics are then used by the analysis tool to provide insights and improvement suggestions. Figure 3.2 shows an overview of the data collection phase.

The type of performance counters to monitor can be specified in our tool. Selecting counters is especially beneficial in scenarios where there is a need to switch among different types of counters to address certain performance issues. For example, to improve the cache utilization and cache footprint of a program, L1, L2, and L3 cache misses or hits seem to be the right counters to monitor. In contrast, in cases where insight about the general performance bottlenecks of a program is important, the CPU utilization counter would be a good fit.

Our tool also supports the option to change the sampling rate, which means changing the frequency at which events are captured. To illustrate further, we look into an example. Assume we are interested in improving cache utilization. L1 and L2 cache misses are the counters we choose to monitor. To gather sufficient samples, we set the sampling rate of our tool to 1/2048. A sampling rate of 1/2048 means that we take 1 example out of every 2048 cache misses that we see. The higher the sampling rate is, the more frequently the tool captures problematic parts of the running program. Consequently, we find more cache misses that have occurred in different program locations.

3.2.2 Attributing and Coalescing Samples

Our profiling tool is meant to be run continuously, in the background, and collect data. As mentioned in section 3.2.1, one key feature of this tool is its space efficiency. During the profile recording phase, we collect samples from all the different programs and store all interesting events across the user space and kernel. By interesting events, we mean the problematic code segments that cause performance reduction, solving which improves the efficiency of a program to a great extent. We choose graph representation to store all this data. This design decision provides us the flexibility to run multiple ad-hoc queries over all the data we collected during a long period. The graph data structure also helps us to create a call graph of the programs and keep track of all the data we collect efficiently. For each sample our tool observes, it captures the Program Counter (PC) of the sample and its metadata. Some examples of the metadata that describe the sample context are:

- Executable's full path at which the sample is taken
- Call stack of the executable
- Thread number

Then, using this metadata our tool creates a call graph of all the calls seen under each executable.

Feature	Description
Binary Name	Selected binary to optimize
Performance counters	The type of events to track
Number of interesting event	The n most frequent events seen
Start time and end time	The interval to examine the binary performance

Table 3.1: List of features that are used for the performance query

After a certain number of events our collection tool observes, it saves all the sample data on disk and clears the main memory. Each file that is written on disk is a record with a graph representation. Each record is a data structure that contains the information of all the executables, i.e. objects, running at the time of taking samples. Each object has a number of nodes that represent different PC addresses in that executable.

3.2.3 Performance Query

We have developed a user-friendly post-processing command-line tool that interacts with users of our tool. It runs queries over all the profiles gathered previously by the collection tool and can filter different features, allowing it to build customized profiles for each scenario. Among the features that can be filtered are shown in Table 3.1.

The ability to run ad-hoc queries enables users to gain detailed insight into their systems. For example, this service helps them to monitor how specific binaries performed in the previous three hours or during the course of a day. It also helps them to identify the most frequent locations in a binary that are not performing as desired.

If our tool has been running in the background for a long time, tens of thousands of files will have been saved on disk. As explained in detail in part 3.2.2, each file is a record of information about the specified performance issues of multiple binaries. To run custom queries over all these data, our tool first scans the files for the given period and loads each one. Then, it merges all the records and builds an aggregated profile. This custom profile

Algorithm 1: Executing Performance Query

```
Input  : Program Name, Counter Name, Start Date, End Date, $N$ Most Frequent
         Events, Collected Profiles
Output: A List of ($PC\ Address, Sample\ Frequency$)
records ← [ ]
graphs ← [ ]
for profile in Collected Profiles do
    if Start Date <=profile.date <= End Date then
        records.append(profile)
    end
end
for record in records do
    filtered_records ← get_info(record, Program  Name, Counter Name)
    graphs.append(filtered_records)
end
aggregated_graph ← merge_graph(graphs)
sort(aggregated_graph, $N$ Most Frequent Events)
return
```

contains the addresses of the samples taken in a binary as well as the total number of samples taken at each location. Ultimately, this information is used by the analysis tool to perform further analysis. Running the performance query is illustrated in Algorithm 1, in which the input is the user query and the output is the aggregated custom profile in the form of a list of {PC addresses, sample frequency, thread number}.

3.3 Profile Analysis and Recommendation Tool

By this point, the tool has collected all the information regarding the particular locations of a binary and the number of events that have occurred. However, the binary-level profile by itself provides no useful insights. For the analysis tool to use all the profiles gathered, it first needs to convert them to source-level profiles. Source-level annotations can then be used to analyze the data further, explore the causes of performance reduction, and recommend solutions.

3.3.1 Data Structure Cache Miss Analysis

In this thesis, our focus is to improve cache utilization by reducing cache misses and cache footprints. In particular, we are interested in finding the structures in which cache misses occur. We extract informative data about the frequency and pattern of cache misses by carefully inspecting the target program. We develop multiple algorithms that analyze the cache miss data and statistics. The output of our analysis algorithms produces recommendations to reorganize those structures, resulting in more efficient cache utilization. The key insight we rely on is that the number of cache misses we get for each field of structures provides us with a good estimation of the field's access pattern and access frequency. This access data is then analyzed to suggest field reordering that will reduce cache misses.

In this thesis, we use multiple analysis algorithms to analyze the problem from different aspects. Each of these algorithms takes into account different sets of information and generates recommendations based on different goals. Two of the main analysis methods we use are cache miss optimization and multi-threaded access optimization. The output of each of these methods includes a set of reordering suggestions that improve the cache utilization of the target program. In addition to producing the reordering suggestions, our tool also estimates the costs of having all these cache misses. Based on the costs of cache misses for different structures, our analysis tool performs cost-benefit analysis on the suggestions provided by each method. It compares and prioritizes the benefits of applying individual suggestions and provides a detailed report on the relative benefits to be gained by reordering.

To annotate each field of structures with the number of cache misses that have occurred in them, first we need to convert the PC addresses to source-level symbols. Typically, to attribute an address in a binary to a specific source-level symbol, we need to look up the DWARF debugging information existing in the binary. Loading all the debug symbols and searching among them can take some time. However, assigning each cache miss sample we observe to a specific field of a structure cannot be done simply by retrieving the debug symbols. Further processing and extensive analysis are needed to take into account all the various scenarios in which the field of a structure is referenced. Some of these cases are as follows:

- Backtracking CPU registers to find a structure and the offset of its field
- Referencing structures as function parameters
- Defining structures as global or local, which changes their location in memory
- Analyzing pointer references to structures in order to find the corresponding offset
- Retrieving a structure with all the information about its field size, type, name, offset, which is necessary for suggestion reordering

3.3.2 Profile Annotation

We use the de-compilation module of Ghidra [7] as an underlying framework and build our annotation algorithm on top of that. To do so, we take the binary of interest as input and de-compile it using Ghidra. During the de-compilation, we retrieve all references to the structures of interest and save them as a compact file on disk. Whenever there is an access to a particular field of a structure in the source code, we call it a reference. These references contain information about all the fields that have been accessed by each

Ordinal Number	Offset	Type	Size(Byte)	Name	Cache Miss Frequency
1	0	long long int	8	a	111
2	8	struct_bar*	8	b	15
3	16	uint64_t	8	c	19
4	24	int	4	d	128

Table 3.2: A sample of a structure annotated by our tool.

particular PC address of a binary. This information is recovered by mapping the assembly-level binary to source code level profile. Since the de-compilation process takes up to several minutes depending on the size of the binary, we do it only once per binary file to save time.

The analysis tool takes two inputs: the references to the structures in the form of a list of {Datatype Name, Datatype Offset, Field Cache-line Number}, and all the gathered profiles in the form of a list of {PC Addresses, Sample Frequency, Thread Number}. By processing this information, the final output is the annotated structures. As illustrated in Algorithm 2, the structure annotation algorithm goes over all PC addresses from which cache miss samples have been taken and catches those that refer to a structure. Once an address that accesses a structure is found, the structure and all the information about its fields are added to a database, and the corresponding field's cache miss number is updated. In this step, any profile samples that are not attributed to a structure are discarded. In the end, all the raw cache miss samples that occurred when structures were being accessed and their thread number are mapped to their corresponding fields. To store fields information, we build a key value store database with a sorted list of the fields as the key and the number of accesses of that sorted list. Table 3.2 shows a sample of a structure annotated by our tool.

Algorithm 2: Profile Annotation

```
Input  : References, Aggregated_Profiles
Output: Annotated Structures
1 datatype_map ← [ ]
2 for pc in Aggregated_Profiles.PC do
3     if References(pc).exists then
4         datatype ← find_datatype(References(pc).datatype_name)
5         datatype_map.add(datatype)
6         update_stats(datatype, References(pc).datatype_offset, pc.frequency)
7     end
8 end
9 return
```

3.3.3 Basic Block Adjustments

To capture the access frequency of structures more accurately, the gathered statistics must be adjusted using basic block information. In fact, huge amounts of hidden cache accesses do not show up in the samples because, once a cache miss happens on one field, the entire cache line is loaded. Consequently, we get no other cache misses on other fields although they are being accessed at the same time. Thus, we have to take into account the cache line information of a structure and extract other accesses information in the same basic block.

Once an access to a structure's field is found at a program location, our algorithm looks into the corresponding basic block and searches for any other accesses to the same cache line of that structure. All the fields that are accessed from the same basic block should have identical frequencies. Thus, our algorithm has to award such fields the same number of cache misses.

In other words, another step is added to the previous structure annotation algorithm 2. Once a PC address that is accessing a field of structure is found, we find its basic block and the maximum PC address in that basic block. Then the same algorithm is repeated for PC addresses now in the range of [the sample PC address: its maximum basic block PC address] and updates their corresponding fields' statistics as well. Algorithm 3 shows the adjusted version of profile annotation algorithm.

To have a better sense of how statistic adjustments for access frequency work, let us look into an example. Consider a sample struct alphabet that has fields a, b, c, and d. Assume a sample program in which both fields a and b of struct Alphabet are accessed 90 percent of the time, leaving 10 percent of the time for accessing the other fields. Assuming that the cache line size is 64 Kb, all fields are on the first cache line. During the collection phase, cache miss samples show up only on the first field, a, although field b has the exact same access frequency. To adjust cache miss frequency, once we have updated cache misses on field a, we look for other accesses from the same basic block that are also on the same cache line, here meaning field b, and award it the same amount.

By scanning the entire basic block of a PC address, we can also gain detailed information about which fields are being accessed together and the order of them. Thus, we take advantage of this feature to annotate every structure with its fields' access pattern and access ordering in addition to access frequency.

By this point, we have extracted all the data we need to feed the analysis algorithms to generate recommendations. We have collected the following information for every structure through the sample collection phase and basic block adjustment:

- Field's access frequency
- Field's access pattern
- Field's access ordering
- Field's access thread number
- The total sum of access frequency per structure

We develop multiple analysis algorithms that leverage this data to provide reordering suggestions based on different goals. The main algorithm we use is cache miss reordering suggestion that considers only access frequency of every field independently. The enhanced version of the cache miss reordering algorithm contains a slightly more complexity. It takes into account access ordering as well as access frequency to make the suggested recommendations match the initial ordering of fields. Finally, the last analysis algorithm we develop is multi-threading reordering suggestion. It separates the fields that are being accessed by different threads and put them into separate cache lines. We explain the analysis algorithms in the following sections in more detail.

3.3.4 Cache Miss Reordering Suggestion

After annotating structures with the cache miss frequencies of their fields, we have two sets of statistics for each structure. One is the initial numbers from mapping all the raw samples we have collected, and two is the adjusted numbers from adding basic block and cache line analysis. Our cache miss reordering suggestion algorithm takes advantage of both of these sets of frequencies to provide a detailed report to users. For suggesting the order of fields within a structure, our algorithm uses the adjusted numbers that consider basic block accesses. However, for computing the benefit of reordering each structure, it uses the initial numbers without considering basic block information.

For generating reordering recommendations within a structure, our algorithm sorts the fields based on their access frequencies. Since the ideal scenario is to avoid randomly ordering the fields with the same or close frequencies, our algorithm also considers fields' access ordering. Thus, in the reordered structure, we group all the highly used fields together and put them in the same cache line. Having all the fields with similar access frequency in the same cache line reduces cache misses. With the first access to any of these fields, the entire cache line is loaded, and subsequent accesses to other fields would become cache hits.

We estimate the amount of benefit gained through reordering by counting the number of cache misses that we save in each structure. To do so, we scan the reordered structure and specify all the fields that are on the same cache line, using the cache line size and each fields' size. We then use the initial cache miss statistics and add the cache miss frequencies on all fields except the first field of each cache line. The result is the estimated number of saved cache misses as the outcome of reordering.

3.3.5 Multithreading Reordering Suggestions

The multi-threading algorithm uses thread numbers and extracts each thread's access pattern to generate reordering suggestions. It takes into account the fields that are accessed by the same thread and tries to keep them together. On the other hand, it separates the fields that are accessed by different threads and spread them into separate cache lines to reduce cache line bouncing. For example, if fields a, e, and h are always being accessed by the same thread, the reasonable order suggestion would be to keep them together although their access frequencies might not be close.

To illustrate more, let us look into another example. Assume structure Alphabet with fields: a, b, and c on the first cache line, and fields i, j, and k on the second cache line. Consider the scenario where fields a, b, and k are always being accessed together by one thread, and fields c, i, and j are being accessed by another thread. In this scenario, the multi-threading algorithm breaks up the fields into separate cache lines, i.e. putting fields a, b, and k in cache line 1 and c, i, and j in cache line 2.

3.3.6 Combining Analysis

Different reordering recommendation algorithms generate different decisions for fixing each structure. We combine these decisions by developing a cost-benefit analysis. To do so, we estimate the cost of each decision and choose the reordering with the minimum cost. In other words, we select the best output out of each analysis on a per-structure basis. To compute the cost of each decision, we calculate how many cache misses will be saved as a result of structure reordering. The number of cache misses provides us a good approximation of how many cycles we might save by fixing the orders. Once the best method for reordering each structure is chosen, we prioritize the order of fixing structures by sorting them based on the maximum benefit. Furthermore, to make it easy for users to apply the reordering suggestions to their code, our tool also outputs its suggestions in C language format.

Algorithm 3: Adjusted Profile Annotation Algorithm

```
Input   : References, Aggregated_Profiles
Output: Annotated Structures
1  datatype_map ← [ ]
2  for pc in Aggregated_Profiles.PC do
3      if References(pc).exists then
4          datatype ← find_datatype(References(pc).datatype_name)
5          datatype_map.add(datatype)
6          update_stats(datatype, References(pc).datatype_offset, pc.frequency)
7          BasicBlock ← find_BasicBlock(pc)
8          for pc_address in [ pc : BasicBlock.maximum_range ] do
9              if References(pc_address).exists then
10                 if References(pc_address).cacheline == References(pc).cacheline
                     then
11                     update_stats(datatype, References(pc_address).datatype_offset,
                         pc.frequency)
12                 end
13             end
14         end
15     end
16 end
17 return
```

Chapter 4

AutoCPA Implementation

We implemented AutoCPA on FreeBSD 12.1 with 3000 source lines of code (SLOC) in C++ for building the collection tool, and 1000 SLOC in Python for the analysis and recommendation tool. We used the hwpmc kernel module and libpmc library in FreeBSD for accessing hardware performance monitoring counters, libprocstat and libkvm for file and process information retrieval from the running kernel, and zlib for data compression.

Our analysis tool uses the Ghidra version 9.1 API to decompile program binaries and relate memory accesses to C language data structures. The Basic Block Model, Code Unit, and Data Type Manager Ghidra APIs available in both Python and Java allowed us to reconstruct data types and retrieve their symbols. To filter specific program addresses that touch the field of a structure, we employed Ghidra's Data Type Reference Finder API. We wrote Ghidra scripts in Python 2.7 and used Ghidra Headless Analyzer tool to run the scripts.

A few parts of the AutoCPA's design are still under development including the multi-threading support and thread analysis. We omitted results from these components.

The way AutoCPA can be used follows this three-step-pattern:

1. Run AutoCPA and run a benchmark to collect and generate representative perfor-

mance data out of a target program.

2. Adjust the target program based on the suggestions provided by AutoCPA.

3. Re-run the benchmark and the target program.

AutoCPA generally is most effective on code that makes numerous accesses to structures and is prone to having many cache misses. Based on real program behavior, AutoCPA detects optimization opportunities in the program that compilers cannot automatically optimize.

Chapter 5

Evaluation

We evaluated AutoCPA's performance by running it against two application: Redis and Memcached. All benchmarks were run on an Intel Xeon E3-1245 v6 running at 3.7 GHz with 8 hyperthreads. Each core has separate 32 KiB data and instruction caches, and a 256 kiB L2 cache. All cores share an 8 MiB L3 cache. In our experiments we monitored the L1 cache misses as the basis for our analysis.

We evaluate the performance improvements primarily using instructions per cycle (IPC). IPC is a measure of the average number of instructions executed in each clock cycle. The final number is the result of dividing the number of instructions by the number of CPU clock cycles computed using high-performance timers [20]. IPC information is obtained from the hardware performance counters and can be monitored for each program running in the system. The number of instructions per cycle is an indicator of how well the program uses the processor. Having this knowledge when an application is running helps us estimate how its throughput changes when we apply AutoCPA's recommendations. For instance, if an application has a higher IPC means that the program is using the processor more efficiently and should be reflected in higher throughput.

We monitored the IPC changes in the target programs and computed the gains we made as a result of applying AutoCPA's recommendations. Our methodology for evaluating the performance of AutoCPA is shown below.

1. Running a test benchmark with each target application
2. Measuring IPC statistics
3. Collecting performance data with AutoCPA's collection tool
4. Getting performance improvement suggestions from AutoCPA's analysis and recommendation tool
5. Applying the changes to the source code of the application
6. Re-running the test benchmark with each target application
7. Re-measuring IPC statistics
8. Comparing the IPCs before and after reordering

Finding a proper workload to use in testing the applications was a challenge as our primary algorithm optimizes structure layout. We need workloads that simulates a real-world scenario generating sufficient cache pressure while using lots of data structures. This eliminates some scientific workloads (e.g., SPLASH) that mostly compute on arrays and matrices of integers and floats.

5.1 Redis

Redis [18] is a popular in-memory key-value store that supports high throughput and low latency. Redis is a challenging application find performance improvements as it is heavily used and under constant development by its many users. Boosting its performance even by few percentage points would result in huge benefits as it is a popular application in large-scale production systems.

We used Redis version 6.0.3 to evaluate our system, and ran the redis-benchmark as the test workload. This benchmarking program simulates an arbitrary number of clients and performing actions on the server, measuring how long it takes for the requests to be completed. Although it can be customized, the Redis benchmark by default runs tests on some of its popular commands. We configured it to run tests on the 13 following most frequently used commands: SET, GET, HSET, INCR, LPOP, LPUSH, LRANGE, MSET, PING, RPOP, RPUSH, SADD, and SPOP.

By default, the Redis benchmark uses a single key. However, to reproduce a real-world-like workload, the -r switch can be used to stress cache misses by means of a large key space. We set the –r switch to 100 million, i.e., the benchmark uses a random key for every operation out of 100 million possible keys. We also configured the benchmark to build 50 parallel connections and to set the payload size to 100 bytes. We ran 2 million requests on the server for each command.

When the server and client benchmark programs ran on the same box, considerable noise appeared in the results. Thus, we instead ran the client benchmark in a different machine to isolate the server. Also using two remote clients, both having the same configuration and running on the same remote machine, increased the load.

After finding the right configuration for the Redis server and benchmark, we ran 13 different experiments (one for each command). We repeated each of the thirteen experiments three times, each time restarting the server to make sure the statistics are reliable and stable. We also ran the entire setting twice for each command: we first ran the original workload and then re-ran it after applying reordering suggestions. In each run, we saved the IPC measurements as a text file, then scanned the text files, extracted the numbers, and took the average and standard deviation of the three runs. Note that the entire experimental process is written in a Python script and is completely automated.

We did not use throughput as our metric because we observed huge amounts of noise and instability in the results. To mitigate the noise and stabilize the statistics, we took the following actions: (I) we pinned the processes to a certain core to prevent the CPU scheduler

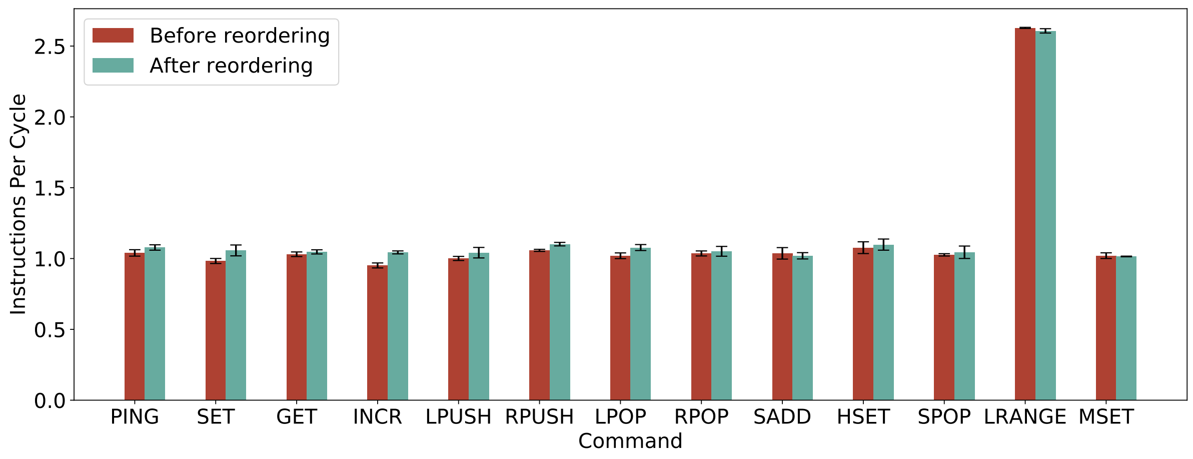

Figure 5.1: IPC changes after reordering the structures for each Redis command. AutoCPA was able to increase the IPC for most of the Redis commands up to 10%.

from moving threads and causing huge variations in the request rates; (II) we disabled disk IO and all storage features; (III) we prevented thermal throttling to maintain performance by changing related CPU settings; (IV) we saturated the CPU load, i.e., running close to 100 % utilization, by adding more clients to increase the workload.

Although each of the above actions helped stabilize the throughput, there was still a considerable amount of noise that was causing inconsistency in the results. Thus, we switched to IPC metric, which ignores all the networking and scheduling problems, resulting in more stable statistics.

We built Redis from its source code and compiled it with the debugging information enabled. The size of the redis-server executable containing debugging information was 4MB. From the 5166 defined data types existing in redis-server executable, we retrieved 388 structures and 32755 references to them in the source code, using DWARF debugging information. We ran the Redis benchmark for 1 hour and collected the performance counter statistics. The counter we monitored in the experiments was L1 cache miss and the sampling intervals was 16384.

Structure Name	Saved Cache Misses
client	10603
redisServer	4101
zlentry	2118
quicklistEntry	1249
quicklistIter	1235
dictht	698
dictEntry	679

Table 5.1: List of Redis structures and the number of cache misses saved on each one as a result of reordering.

In one hour, we collected 7 MB of data, in 110 files, each of 70 KB. Aggregating all these profiles provided us with 5240 unique PC addresses as well as their corresponding cache miss frequencies in the target program that our collection tool observed. We then fed our analysis tool with the references and aggregated profile to get reordering recommendations.

Note that the number of cache miss samples collected by our tool represents only a subset of all the misses that occurred in the system. Thus, the actual number of cache misses equals the result of multiplying the sample cache misses by the sampling intervals.

According to the recommendations provided by our tool, *client*, *redisServer*, and *zlentry* were the top three structures that provide the most benefit as the result of a potential reordering. We reordered the structures shown in Table 5.1 based on AutoCPA's recommended output, reordering which would have saved more than 500 cache misses.

Figure 5.1 shows the IPC changes for each Redis command after the structures were reordered. As shown in the graph, AutoCPA increased the IPC for most of the Redis commands by up to 10%.

5.2 Memcached

Memcached [17] is a popular high-performance distributed memory-caching system, which is used for accelerating web applications. Its simple design and easy-to-use features make it popular caching system. Similar to Redis, it also heavily developed and any optimization benefit would be substantial.

We used Memcached version 1.6.6 built with debug symbols enabled, resulting in a 660 KiB Memcached server executable. Memcached was configured to use 4 threads for processing incoming requests. We collected 6640 different references to structures from analyzing the Memcached executable.

We used the Mutilate benchmark [10], which is a Memcached load generator designed to simulate real world workloads. Unlike Redis, Memcached offers no option to run a benchmark for individual commands. Thus, we only get statistics for one run of Memcached's overall performance.

Mutilate was configured it to simulate a realistic client workload. It was set to connect to the localhost, to establish 50 connections, and to run 5 million queries per second as the target. We ran the Mutilate benchmark on the Memcached server for half an hour and collected the performance counter statistics. The counter we monitored in the experiments was L1 cache miss, and the sampling interval was 16384. We collected 2 MB of data containing 30 files, each 70 KB in size. Aggregating this data provided us with the 2349 unique PC addresses in which our collection tool observed cache misses.

We configured the Memcached server and Mutilate benchmark, and ran the entire workload twice: once for the original workload and again for the workload after applying reordering suggestions. For each setting, we repeated the experiments three times and saved the IPC measurements as text files. We then scanned the text files, extracted the numbers, and took the average and standard deviations of the three runs.

According to the AutoCPA's analysis, *stritem*, *conn*, and *mc_resp* were the top three structures that provide the most benefit as the result of a potential reordering. We fixed

Structure Name	Saved Cache Misses
conn	1308
_mc_resp	945
Settings	826
LIBEVENT_THREAD	514
Bipbuf_t	354
Token_s	165
Cache_t	83
_lru_bump_buf	67
Thread_stats	52

Table 5.2: List of Memcached structures and the number of cache misses saved on each one as a result of reordering.

the structures shown in Table 5.2, reordering which would have saved more than 50 cache misses:

The first recommended structure was not reordered, although about 40 percent of cache misses were attributed to it, because in Memcached's codebase, some structures are cast into one another. We could not reorder such structures since doing so breaks compatibility. Fixing this problem requires some manual inspection of the source code.

Structure *stritem* is an example of structures that are cast into one another. To reorder such structures automatically, we first have to find other similar structures with the same layout and be consistent with the reordering of all of them. Thus, after finding all the related structures, we have to keep the order of the mutual fields the same and reorder the remaining fields accordingly. Since our analysis algorithm does not take into account casting, we reordered all structures except the one that was cast.

When comparing the results before and after reordering, we noticed that the IPC improvement is not considerable, i.e. less than 1 percent. This result was expected since we missed some of the crucial cache miss data due to the structure casting done in Mem-

IPC Improvement	Cache Miss Reduction
0.8%	7.1%

Table 5.3: Changes in the IPC and number of cache misses, as a result of reordering Memcached structures.

cached's source code. Instead, we used another metric, obtained from the PMC tool of FreeBSD, to assess the impact of AutoCPA on Memcached; that is, for each run, we monitored the total cache misses that occurred. According to Table 5.3, reordering all suggested structures except the first one resulted in 7.1 percent of cache miss reduction.

Chapter 6

Conclusion and Discussion

This thesis introduces AutoCPA, a fully-automated low-overhead profiler that makes high level recommends structure reordering to improve cache efficiency in applications. AutoCPA uses modern performance counters to collect program-wide profiles, which are then analyzed to extract memory access patterns and frequencies. The recommendations provided by AutoCPA, are the results of its cost-benefit analysis over several cache optimization algorithms. The tool is easy to use, and no specific setup is needed for the target system. It works on highly optimized executables and is independent from compilers. The system's evaluation on both Redis and Memcached shows the effectiveness of AutoCPA to improve applications' performance, with up to 10% IPC improvement in Redis commands and 7.1% cache miss reduction in Memcached.

Several opportunities exist for extending the usefulness of AutoCPA:

- The current tool's sampling rate can be changed to a higher or lower frequency, but this setting is fixed throughout the entire profile collection process. In future work, a dynamic sampling can be implemented to improve the trade-off between profile resolution and system overhead.
- The multi-threaded analysis is still incomplete, and we require a few improvements

to support the thread implementation to identify false sharing and to reduce core-to-core transfers.

- Although the access pattern and access frequency of fields are analyzed to generate an optimized structure layout, further parameters such as data alignment and structure padding can be considered to improve field reordering.
- The tool can be made more generic by building more analysis passes into it that take into account multiple performance counters such as L2 misses, L3 misses, or branch mispredictions to provide additional optimization opportunities.
- Finally, we need to evaluate AutoCPA on more applications such as web servers, and OS kernel to better assess its effectiveness.

The source code for AutoCPA is available at `https://github.com/rcslab/AutoCPA/`.

References

[1] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, November 1997.

[3] Gautam Chakrabarti, Fred Chow, and L PathScale. Structure layout optimizations in the open64 compiler: Design, implementation and measurements. In *Open64 Workshop at the International Symposium on Code Generation and Optimization*, 2008.

[4] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, 1999.

[5] Thomas M Conte, Burzin A Patel, Kishore N Menezes, and J Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, 1996.

[6] Jeffrey Dean, James E Hicks, Carl A Waldspurger, William E Weihl, and George Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order

processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 292–302. IEEE, 1997.

[7] NSA's Research Directorate. Ghidra: A software reverse engineering (SRE) suite of tools. https://ghidra-sre.org/.

[8] Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in gcc. In *Proceedings of the GCC Developers' Summit*, pages 69–92. Citeseer, 2005.

[9] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. Practical structure layout optimization and advice. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[10] Jacob Leverich. Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate, Jan 2017.

[11] Jin Lin and Pen-Chung Yew. A compiler framework for general memory layout optimizations targeting structures. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, pages 1–8, 2010.

[12] Matthew C Merten, Andrew R Trick, Christopher N George, John C Gyllenhaal, and Wen-mei W Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 136–147, 1999.

[13] P Pirkelbauer, P Lin, T Vanderbruggen, and C Liao. Xplacer: Automatic analysis of cpu/gpu access patterns. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019.

[14] N Prashantha, T Vikram, and N Vaivaswatha. Implementing data layout optimizations in the llvm framework. 2014.

[15] Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. Structure layout optimization for multithreaded programs. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 271–282. IEEE, 2007.

[16] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 30(4):65–79, July 2010.

[17] The Memcached Team. Memcached: A distributed memory object caching system. http://memcached.org/, May 2016.

[18] The Redis Team. Redis: An Open Source, In-memory Data Structure Store. https://redis.io/.

[19] Omri Traub, Stuart Schechter, and Michael D Smith. Ephemeral instrumentation for lightweight program profiling. *Unpublished technical report, Department of Electrical Engineering and Computer Science, Hardward University, Cambridge, Massachusetts*, 2000.

[20] Wikipedia Foundation, Inc. Instruction Per Cycle. https://en.wikipedia.org/wiki/Instructions_per_cycle.

[21] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: a higher order theory of locality. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 343–356, 2013.

[22] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J Bradley Chen, and Michael D Smith. System support for automatic profiling and optimization. *ACM SIGOPS Operating Systems Review*, 31(5):15–26, 1997.

[23] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. *ACM SIGPLAN Notices*, 39(6):255–266, 2004.