Karlsruher Institut für Technologie

Call Graph Based Instruction Prefetching on Precompiled Executables

Bachelorarbeit
von

Joshua Bachmeier

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Mathias Gottschlag, M. Sc.

Bearbeitungszeit: 5. Mai 2017 – 4. September 2017

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 4. September 2017

Abstract

The performance of scale-out and online transaction processing workloads, often found in modern datacenters, suffer from high instruction cache miss rates. These are caused by large instruction working sets and irregular, non-sequential execution flow patterns that are typical for these applications. Past research has suggested many approaches to reduce instruction cache miss rates. However, most existing solutions are either implemented in hardware or require special compiler or operating system support.

In this work, we present *Call Graph Prefetching on Precompiled Executables (CGPoPE)* as a system to reduce instruction cache misses caused by function calls. CGPoPE builds the call graph of an application by intercepting and recording function calls. It then extracts the most frequent callee of each function and injects prefetch instructions into the ordinarily compiled binary of the application. The mechanism we use to inject prefetch instructions can be applied to insert arbitrary instructions into the machine code of an application. This design enables the implementation of live patching via the operating system, which is able to perform the binary augmentation automatically when restarting an application.

We evaluate CGPoPE using a prototypical implementation for ELF files and the x86 architecture. Although we find that CGPoPE does not improve the performance of our benchmarks, we show that all injected prefetches are effective. Our analysis also reveals that the injection of instructions into unmodified binaries does not cause significant additional run time overhead, and that the effectivity of CGPoPE can likely be increased by more aggressive prefetching.

Deutsche Zusammenfassung

Die Performanz von Scale-Out und Online-Transaktionsverarbeitungs-Workloads, wie sie häufig in modernen Datenzentren zu finden sind, wird durch hohe Miss-Raten in Instruktions-Caches beeinträchtigt. Diese Misses werden durch große Instruktions-Working-Sets und irreguläre, nicht-sequentielle Ausführungsflussmuster verursacht, wie sie typisch für diese Art von Anwendung sind. In vorangegangenen Arbeiten wurden viele Herangehensweisen zur Reduktion von Instruktions-Cache-Miss-Raten verfolgt, die allerdings häufig entweder in Hardware implementiert sind oder spezielle Unterstützung durch den Compiler oder das Betriebssystem benötigen.

In dieser Arbeit präsentieren wir *Call Graph Prefetching on Precompiled Executables (CGPoPE)*, ein System zur Reduktion von Instruktions-Cache-Misses, die durch Funktionsaufrufe verursacht werden. CGPoPE generiert den Aufrufsgraphen einer Anwendung, indem es Funktionsaufrufe abfängt und aufnimmt. Für jede Funktion extrahiert es dann die am häufigsten aufgerufene Kindfunktion und fügt Prefetch-Instruktion in die gewöhnlich kompilierte Binary der Anwendung ein. Dieses Design ermöglicht die Implementierung von Live-Patching von Programmen durch das Betriebsystem, da dieses in der Lage ist, die Modifikation der Binaries automatisch beim Neustart der Anwendung durchzuführen.

Wir evaluieren CGPoPE mittels einer prototypischen Implementierung für ELF-Dateien auf der x86-Architektur. Während wir feststellen, dass CGPoPE die Performanz unserer Benchmarks nicht signifikant erhöht, zeigen wir dennoch, dass alle eingefügten Prefetches effektiv sind. Unsere Analyse offenbart auch, dass das Einfügen von Instruktionen in unmodifizierte Binaries keinen nennenswerten zusätzlichen Laufzeitoverhead verursacht, und dass die Effektivität von CGPoPE vermutlich durch aggressiveres Prefetching erhöht werden kann.

Contents

Chapter 1

Introduction

The primary performance bottleneck of database management systems (DBMSs) has traditionally been input/output (I/O) operation, caused by long disk access latencies [4]. However, due to the availability of larger and cheaper main memory, more data can be stored in memory [5]. Whereas performance of database applications has improved through this, the upper bound for the performance shifts to memory access time [4] and thus cache effects have become more relevant.

The performance of processors has outpaced that of main memory since as early as the 1980s [6, 26]. From then on, this gap continued to grow and only recently, with processor technology reaching its upper bound for sequential execution speed [35], the growth has begun to cease. Caches can help to fill this gap, by bringing frequently used memory content closer to the processor [6]. However, research has shown that scale-out and especially online transaction processing (OLTP) workloads suffer from high data and instruction cache miss rates [10].

Research has already explored possible reasons for this. Kanev et al. observed that instruction working set sizes of typical datacenter applications can significantly exceed the size of the level 2 cache (L2) in common architectures [18]. Moreover, cloud applications and DBMSs exhibit complex, non-sequential call patterns, due to their often modular and high-level design [4, 10].

In order to assert effective utilisation of available instruction cache capacities, a more intelligent policy is required. Most CPUs provide a simple next-N-line (NL) prefetcher, that prefetches the next N cache lines following the program counter (PC). As the NL prefetcher is only effective for instructions within this *prefetch window*, the policy especially needs to capture branches out of the window. For modularly designed software with relative small functions, which includes DBMSs and cloud applications, the majority of these long jumps are function calls, because the NL prefetcher already covers the whole function body in most cases.

Most existing prefetching schemes that capture complex execution patterns are either implemented in hardware or, in case of software prefetching schemes, inject prefetch instructions during compilation. Hardware prefetchers that are available on commodity hardware are usually limited to NL prefetchers, due to the long development cycle of computer hardware. Thus, to effectively deploy a more intelligent prefetching scheme, a software implemented approach is required. The disadvantage of most existing software prefetchers is that they require to recompile the application to inject prefetch instructions. A solution that does not rely on compiler support would be applicable existing legacy or proprietary applications, and would allow to live patch applications on a running system, only requiring the application to be restarted.

Based on the work of Annavaram, Patel and Davidson [4] we developed *Call Graph Prefetching on Precompiled Executables (CGPoPE)*, a binary augmenter that optimises an application to reduce instruction cache miss rates based on its call graph. The augmenter operates on ordinarily compiled executables, thus evading the need for special run time or compiler support.

CGPoPE first monitors the program while it executes a representative sample workload. We use the dynamic instrumentation and monitoring tool SystemTap [36] to trace and record function calls made by the program. By accumulating the recorded function calls and counting the occurrences of each caller-callee pair, a call graph annotated with call frequencies can be built. CGPoPE uses this graph to inject prefetch instructions into each function, prefetching the most frequent callee of that function. We show that the injection of arbitrary instructions into ordinarily compiled binaries is possible by hijacking the execution flow of functions via the insertion of jumps. However, we find that CGPoPE does not significantly reduce the overall instruction cache miss rates in its current form and thus does not improve the performance of our benchmarks. Nontheless, we show that all prefetches issued by CGPoPE are effective, and that no significant additional overhead is caused by hijacking the functions. We conclude that the effectivity of CGPoPE can be improved by making adaptions to the prefetching policy while still using the presented function hijacking mechanism.

In Chapter 2 we review existing approaches to reduce instruction cache miss rates, and compare them to CGPoPE. Chapter 3 discusses the design of CGPoPE, with respect to conditions and requirements imposed across different architectures and technologies. Then, we present the details of our implementation in Chapter 4, concertising the given design of CGPoPE to 64-bit binaries in the executable and linkable format (ELF) [38] for the x86 architecture [8]. In Chapter 5 we evaluate our implementation. We also give a detailed analysis on a smaller scale (microbenchmark), to both outline the cases in which CGPoPE produces effective prefetches and identify likely reasons why our system fails to increase the performance of software with high instruction cache miss rates significantly. In Chapter 6 we conclude on how CGPoPE can help to reduce instruction cache miss rates and propose future directions to mitigate the observed issues.

Chapter 2

Background & Related Work

In this work, we present *Call Graph Prefetching on Precompiled Executables (CGPoPE)*, an instruction prefetching mechanism based on Call Graph Prefetching (CGP) by Annavaram, Patel and Davidson [4]. In this chapter we cover analysis of instruction cache performance bottlenecks and review existing approaches to mitigate these.

Performance bottlenecks in database management systems (DBMSs) and cloud applications have been pinpointed to instruction cache inefficiencies in the past. We start with a brief presentation of the most common hardware prefetching scheme already available in many modern processors in Section 2.1. In Section 2.2 we summarise research and studies that analyse common workloads and identifies high instruction cache miss rates as a major performance bottleneck in these applications. In Section 2.3 we review existing approaches to reduce instruction cache miss rates. We compare these to our approach with regard to both the *mechanism* by which the prefetch is performed, and the *policy* by which the prefetch target is selected.

2.1 Next-N-Line Prefetching

As most existing prefetching schemes cooperate with or supplement existing hardware prefetchers, we briefly present the most common hardware prefetcher, next-N-line (NL) prefetching, to give a basis for further review of existing approaches. NL prefetching is a hardware-implemented prefetching mechanism based on next-line prefetching [32]. As only the minority of instructions are branch instructions, the sequentially next instruction in the program is most likely the one to be executed next. Next-line prefetching takes advantage of this observation, by always prefetching the next instruction line. NL prefetching is an extension to this, which does not only

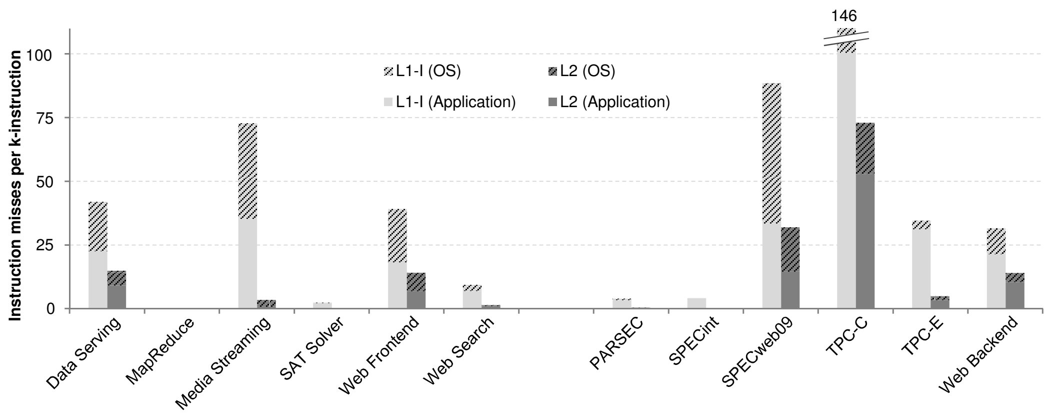


Figure 2.1: "L1-I and L2 instruction cache miss rates for scale-out workloads (left) and traditional benchmarks (right)" [10]

prefetches the next N lines, instead of only a single line. Many modern CPUs provide a builtin hardware NL prefetcher. However, as NL prefetchers only cover simple, sequential execution patterns, they are insufficient to effectively prefetch more complex execution flows, such as branches and function calls.

2.2 Performance Bottlenecks in DBMSs and Cloud Applications

Ferdman et al. studied inefficiencies in processor micro-architecture when running scale-out workloads, and compared these to traditional online transaction processing (OLTP) workloads [10]. They identified high instruction cache miss rates as one of the major performance bottlenecks in scale-out and especially OLTP workloads. Their findings are depicted in Figure 2.1. The diagram shows the instruction cache misses per 1000 instructions for selected scale-out workloads on the left, and OLTP workloads on the right. Instruction cache miss rates are highest for the level 1 instruction cache (L1-I), but the problem is also present for the level 2 cache (L2), most notably for OLTP workloads like TPC-C [39]. For most workloads, the majority of instruction cache misses occurs while running in user mode (marked as "Application" in the figure), rather than in kernel mode (marked as "OS"), suggesting the application as a promising target for optimisation efforts.

Ferdman et al. suggest the complexity of modern applications as a likely reason for the poor utilisation of instruction caches: Since these are often written in high-level languages, use external libraries and frequently trap into the kernel, they exhibit complex, non-sequential call patterns, which are

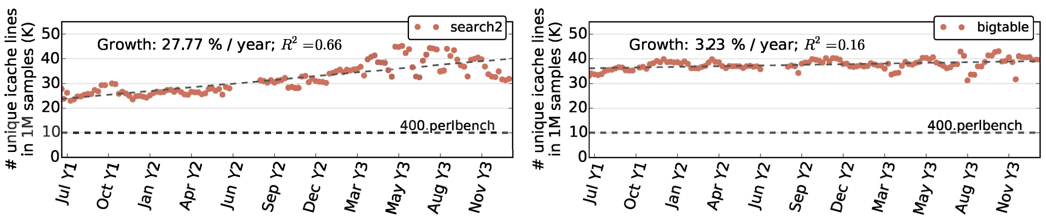


Figure 2.2: “Large instruction cache footprints” [18]

not captured by the CPU’s NL prefetcher. Their argument targets cloud applications, but can be applied to DBMSs as well, due to their often similar structure [4].

Kanev et al. performed a live analysis of datacenter jobs over a period of three years [18]. By extending their analysis over multiple years, they identified characteristics and trends in datacenter applications. They confirm the findings of Ferdman et al., that L2 instruction cache miss rates are usually high, and even find the to be up to 50 % higher in their measurements.

The paper also offers further insight into the causes of this phenomenon. By counting the number of unique cache lines in recorded samples, they give a measure of a program’s instruction working set size. Figure 2.2 shows the result of applying this approach over a period of 30 months. Not only did they find that instruction working set sizes can be 688 KiB or more, and thus exceed the size of the L2 in common architectures (Intel: 256 KiB, AMD: 512 KiB), but the figure also shows that the instruction working sets are trending to grow even larger, with growth rates ranging from 3.23 to 27.77 % per year for some applications.

2.3 Mitigating Instruction Cache Bottlenecks

Many approaches have been proposed by the research community to reduce the instruction cache bottleneck. We review the approaches regarding the prefetching policy which they implement and the mechanism by which they issue prefetches.

Policy. As traditional hardware based NL prefetchers already effectively covers sequential execution [31], the focus of most research is on the mitigation of instruction misses caused by interruptions of these sequences, i.e. branches and jumps. To address this, many approaches issue instruction prefetches based on the branch behaviour of the program. A straight-forward method for this is proposed by Hsu and Smith [15]. Their employ a *target* *prediction* table, which for each instruction cache line stores the line that most recently followed the active line. Based on this and the current program counter (PC), a likely next instruction line can be prefetched. This approach implements a standalone prefetching mechanism, completely replacing the NL prefetcher. However, as the majority of code is sequential, the entry in the target prediction table will simply be the next instruction line (i.e. $\mathrm{PC} + 1$) for most instructions. As this common case is also covered by much simpler NL prefetchers, a solution that focuses on non-sequential execution is more suitable. CGPoPE issues prefetches of non-sequential function calls in addition to any underlying hardware prefetcher, thus taking advantage of NL prefetching for sequential execution.

Shyamala et al. present Basicblock Instruction Prefetching (BIP), which is limited to non-sequential execution flow transitions. *Basic blocks* are blocks of machine code consisting solely of sequential instructions, which are then interconnected by branch instructions, such as the two arms of an `if`/`else` clause or the body of a loop construct [30]. BIP builds a graph of basic blocks and approximates transition probabilities between these blocks in a learning phase. Based on these probabilities, BIP issues prefetches of likely target basic blocks. Similarly, Spracklen, Chou and Abraham make use of the effectivity of the already available NL prefetcher and also only handle cases that disrupt the sequential execution flow, which they call *discontinuities* [33]. They monitor and record these discontinuities in a small hardware table and prefetch the target of each discontinuity just before it comes up next. Although both of these approaches limit the additional prefetching effort to non-sequential accesses, Luk and Mowry observe that the NL prefetcher already covers many non-sequential accesses, namely low-distance control-flow branches that fall within the next N lines [22]. Consequently, it is sufficient to prefetch targets of branches out of this window, rather than all discontinuities. This observation is considered by CGPoPE, which only prefetches targets of function calls, and thus relies on NL prefetching for all intra-function execution.

The same observation is exploited by Annavaram, Patel and Davidson in CGP. They build an *annotated call graph* of the program, which stores for each function the functions called during its first execution, and their invocation order [4]. Based on this, they issue instruction prefetches for the next callee at the beginning of the function and upon each successive return. The policy employed by CGPoPE is based on CGP. We also utilise a call graph to identify likely callees for each function. But instead of annotating our call graph with the order in which the callees were called during the first run of the function, we weigh each edge with the frequency with which the call was performed over multiple executions. This is a more expressive profile, as it includes not only the first run of the function, but an average over all invocations, has however the disadvantage of not including the order in which the functions where called.

A different policy, which is not directly based on the branch/jump behaviour of a program, is employed in RAS Directed Instruction Prefetching (RDIP), developed by Kolli, Saidi and Wenisch [20]. They observe a correlation between program context and cache miss patterns. The former can be identified by the current state of the return address stack (RAS). RDIP records instruction cache misses that occurred while running in each context and prefetches these instructions just before the same context is visited again. Whereas this approach shows similarities to a call graph based policy in that a function that is often called from a given context is likely prefetched by RDIP, it comes at the disadvantage of having to be retrained every time the application is started: Only on repeated visits of specific contexts will the prefetching be effective.

This is a disadvantage that RDIP has in common with many hardware implemented approaches: These often require live training time to become effective. CGPoPE on the other hand, only requires one profiling run to build the call graph, after which any subsequent executions of the application are optimised from the beginning.

Mechanism. In order to asses the advantages of a software based approach like CGPoPE, we review existing prefetching schemes with regard to the mechanism used. Most implementations prefetch target instructions in hardware [4, 15, 20, 33, 30] or issue prefetches in software, directed by the compiler [22, 28]. However, there are also approaches that are not based on prefetching, but instead improve instruction cache patterns through scheduling techniques [14, 42].

The advantage of hardware based approaches is that they have very limited run time overhead, as they can be implemented as a separate hardware unit operating largely independent of the CPU cores. But since hardware support is also very expensive and has long development cycles, it is usually unavailable on production systems. Precisely over this, software based approaches like CGPoPE have the advantage, because they are much easier and cheaper to deploy. Additionally, software systems have more high-level run time information immediately available and can issue according prefetches from the beginning, while hardware prefetchers typically require initial training time to detect an access pattern that can be prefetched [21].

Existing software based prefetching systems are implemented in the compiler, which has the disadvantage of either limiting the prefetching policy to a static analysis of the program, or requiring to recompile the program after it has been run and the profiling data is collected. This is especially a drawback in environments where recompilation is not an option or very cumbersome, for example when working with proprietary or legacy software, or when live patching applications on a running system.

CGPoPE uses a software based prefetching mechanism, but rather than inserting prefetch instructions during compilation, we augment precompiled executables, by injecting prefetch instructions directly into the machine code. This way we include dynamically generated profiling information, while eliminating the step of recompiling the program with the training data as an additional input. Instead, the existing binary can be "patched" with prefetch instructions.

Annavaram, Patel and Davidson present both a hardware and a software implementation of CGP [4]. When compared to CGPoPE, especially the software implementation is relevant: They also insert prefetch instructions into the program, though they do not specify the method of inserting the instructions. As we discuss in Chapter 3, this is a very complex problem, especially when refraining from compiler-directed insertion. In this work, we apply an adapted and simplified version of the policy of CGP to precompiled executables. Building on CGP, we present a method to insert prefetch instructions into a program.

An entirely different approach to reduce instruction cache miss rates is followed by Harizopoulos and Ailamaki and Zhou and Ross [14, 42]. Their approaches are not based on prefetching instructions likely executed in the near future, but direct a system to execute multiple instances of the same code locally and temporally close to each other, to maximise reusing of instruction cache lines.

Harizopoulos and Ailamaki developed Synchronised Threads Through Explicit Processor Scheduling (STEPS), a scheduling technique to improve instruction cache performance of multithreaded applications. STEPS groups threads into *teams*, that run the same system component and thus share code. Within a team, STEPS schedules context switches to maximise instruction cache line reuse [14].

Zhou and Ross use knowledge about shared code in an application to increase reusing of instruction cache lines. They observe high rates of instruction cache thrashing between different database operations, resulting in the working set of a specific operation no longer being present in the instruction cache once it is executed again. They achieve higher cache hit rates, by buffering and reordering database operations, so that the same operations are executed in sequence [42].

These high-level scheduling approaches have an advantage in common with hardware based mechanisms: They can include run time information live into the optimisation, while at the same time being much more flexible. Their disadvantage is that they have higher overhead, as the thread scheduling and operator reordering needs to be managed at run time from within the operating system and the DBMS driver, respectively. The execution of operating system and DBMS driver code increases the instruction working set of the application, further straining the instruction cache. CGPoPE injects the prefetch code directly into the application executable, thereby inserting as few additional instructions as possible. Whereas this makes the prefetching policy static (the prefetch targets are encoded into the executable prior to run time), it also minimises the additional run time and especially instruction cache overhead.

Chapter 3

Design

In this work, we introduce *Call Graph Prefetching on Precompiled Executables (CGPoPE)*, a system for reducing instruction cache miss rates in database management systems (DBMSs). For this purpose, CGPoPE analyses the call graph of a program and instruments each function to prefetch its most frequent callee. Contrary to most existing software prefetching schemes, our solution operates on ordinarily compiled binaries, which it patches by injecting prefetch instructions.

In this chapter, we present the design of CGPoPE. Initially, Section 3.1 provides a brief overview of CGPoPE's design. Then, we show how the call graph is generated in Section 3.2. In Section 3.3 we explore techniques to inject prefetch instructions into a binary based on a given call graph.

3.1 Overview

CGPoPE operates in three stages, as is illustrated in Figure 3.1. First, it monitors the application and records and counts function calls. Second, the

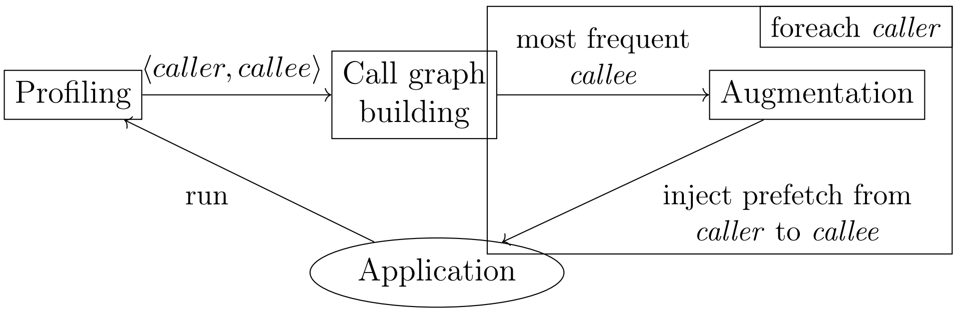


Figure 3.1: Overview of CGPoPE

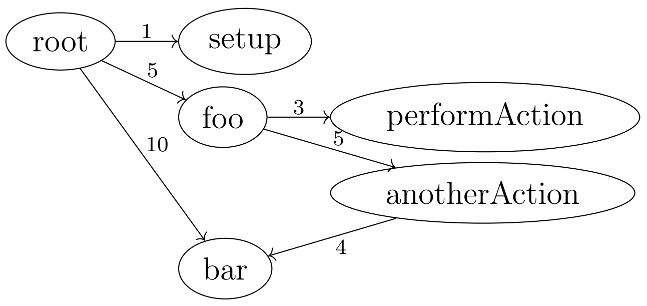


Figure 3.2: Abstract representation of a weighted call graph

call graph of the program is derived from these records. It stores for each function which other functions it calls, and how frequently. Third, CGPoPE reads the most frequent callee of each function from the call graph and for each of these ⟨*caller, callee*⟩ pairs, CGPoPE instruments *caller* to prefetch *callee*.

The main problem of the first stage is to intercept all function calls made by the program, without altering its source code or executable. The third stage, the injection of prefetch code, is particularly difficult, with the insertion of arbitrary instructions into binaries being the major challenge.

3.2 Call Graph Generation

To find suitable prefetch targets we use a call graph. In this section we discuss how CGPoPE generates the call graph and the data structures used to store it.

In Section 3.2.1 we provide a formal definition of a call graph. We then present the generation process of such a call graph in Section 3.2.2. In Section 3.2.3 we demonstrate how the generated call graph can be used to identify suitable prefetch targets.

3.2.1 Call Graph

The *call graph* of a program is defined as the directed graph containing one *vertex* for each function of the program [17]. An *edge* between to vertices A and B exists if and only if the function A calls B. We also define the *weighted call graph.* The weighted call graph additionally associates a weight with each edge $\langle A, B\rangle$, representing how frequent A calls B.

Figure 3.2 shows an example of such a weighted call graph. In the example, the function *root* calls *setup* one time, *foo* five and *bar* ten times.

The function *foo* calls *performAction* three and *anotherAction* five times, which in turn calls *bar* four times.

Note that the call graph generated by CGPoPE only contains vertices and edges for calls that actually occurred during the profile run, not for all hypothetical calls that are contained in the program.

3.2.2 Call Tracing

The process of generating the call graph for a given program can be divided into two stages, *profiling* and *accumulation.*

Profiling. In the profiling stage, CGPoPE monitors the program while it executes a representative sample workload. The binary of the program does not have to be modified for this.

CGPoPE intercepts function calls performed by the monitored process. Our solution uses kernel mode software tracing and does not rely on hardware tracing features, such as event counters. The advantage of this is that CGPoPE can also work on CPUs which do not provide a *precise* event counter for function call events (which includes most Intel and AMD CPUs [16, 3]). The interception is done by instrumenting the process to trap into the kernel in the event of a function call. By examining the state of the process at the moment it was interrupted, the *caller* and the *callee* of the call can be determined. The event is then recorded in the form of a ⟨*caller, callee*⟩ tuple. For the example from Figure 3.2, there are be 28 such records, the sum of the weights of all edges.

Accumulation. In the accumulation stage, the recorded calls are accumulated into a call graph. The call graph is represented in the form of ⟨*caller, callee, frequency*⟩ tuples, where *frequency* designates the number of recorded calls from that caller to that callee, over the total time of the profiling run. Each of these tuples can then be seen as an edge in the weighted call graph, the caller being the source and the callee the destination vertex. Accordingly, frequency is then the weight of that edge. Note that by this accumulation step, any information on the order the calls occurred is lost. Only the call frequencies are retained.

To accumulate the record tuples, CGPoPE uses a hash table. Each ⟨*caller, callee*⟩ pair is mapped to its *frequency.* The hash table representation of a call graph is equivalent to its 3-tuple representation. CGPoPE processes the recorded function call event tuples and increments the corresponding

Key		*Value*
Caller	Callee	Frequency
root	setup	1
root	foo	5
root	bar	10
foo	performAction	3
foo	anotherAction	5
anotherAction	bar	4

Table 3.1: Hash table/3-tuple representation of a weighted call graph

entry in the hash table by 1, if one exits. If no such entry exits, it is created and initialised to 1.

Table 3.1 shows the hash table representation of the call graph from Figure 3.2. The first two columns designate the caller and the callee, i.e. the key of this entry in the hash table, while the third columns gives the value of the entry and thus the frequency of that call. The table also illustrates the equivalence of the 3-tuple and the hash table representation: Each line can be seen as a hash table entry as well as a tuple.

The two stages, *profiling* and *accumulation*, are executed in parallel. As noted in Chapter 1, online transaction processing (OLTP) and scale-out workloads are often characterised by consisting of many small functions. CGPoPE creates a distinct record for each function call in the profiling stage, so the amount of data produced in this stage might be considerable. After the accumulation stage however, the amount is much more manageable, because multiple records for the same call are merged into one. Also, since now there is at most one record for every possible call, the number of records is in $\mathcal{O}(n^2)$, n being the total number of functions in the program. The size of the profiling data is hence bounded statically by the complexity and size of the program, rather then by the complexity and length of the profiling run. By performing the accumulation live, each 2-tuple can be inserted into the hash table upon creation and does not have to be stored. The insertion into the hash table can be performed in $\mathcal{O}(1)$ [23], so the additional overhead imposed on the profiling stage is minimal.

The call graph is then stored persistently in its 3-tuple representation, so it can be processed in the next step to perform the actual augmentation of the binary.

3.2.3 Prefetch Target Selection

We can now select a suitable prefetch target for each function. For reasons of simplicity, we limit our solution to injecting a single prefetch per function. A suitable prefetch target is the callee function which is prone to cause the most instruction cache misses when calling it. A natural candidate for this is the most frequent callee.

The most frequent callees of all functions can be extracted from the weighted call graph as follows. For each vertex, select the highest weighted outgoing edge. Remove all other outgoing edges from that vertex. Remove all vertices that are separated from the remainder of the graph. The weights of the remaining edges are also no longer required. The resulting *filtered call graph* contains only one outgoing edge per vertex, and the set of all edges gives the prefetch operations to be injected: For each edge $\langle \mathit{caller}, \mathit{callee} \rangle$, CGPoPE injects a prefetch instruction for *callee* into *caller*.

3.3 Binary augmentation

Once suitable prefetch targets are selected, the binary can be augmented with the corresponding prefetch instructions. Instructions cannot be simply inserted at arbitrary points in the program. In Section 3.3.1 we show why and develop function hijacking, a mechanism to inject arbitrary instructions into an unmodified binary. In Section 3.3.2 we then apply this mechanism to the profiling data from Section 3.2.

3.3.1 Instruction Insertion

As part of the design of CGPoPE we need to insert prefetch instructions into unmodified binaries, which poses a significant problem. A fundamental restriction of memory-based computing is that in a readily compiled binary, one cannot simply insert a new instruction at a given point by *shifting* all following instruction to make space. This is due to the fact that the code might contain references to the shifted instructions in the form of jumps. Jump instructions refer to a location in the programs address space (AS) as the jump target. When this target lies in the shifted segment, its address is changed, and at its original position (to which the jump instruction refers), a different instruction is found. The text section is also often directly followed by data sections that contain global variables, references to which would also be broken. As these references might be calculated in an arbitrary complex way by the program, they cannot be detected unless special information is provided by the compiler.

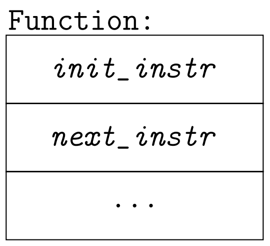
To solve this problem we consider three distinct approaches: (1) the usage of relocation tables, (2) requiring position independent code (PIC) for augmentation or (3) putting the new instructions at the end of the program's memory image and redirecting the execution flow there. The first two approaches attempt to enable the shifting processes, while the third bypasses it. As we show, only the third approach is viable to our problem. All three approaches are described in terms of inserting a single new instruction at the beginning of a function, the argumentation in each case is, however, also applicable to the problem of inserting multiple instructions at an arbitrary position in the binary.

(1) Relocation Tables. For the purpose of linking, compilers generate relocation tables. The tables' entries describe positions in the object files which contain references to symbols. A symbol in the context of programs and object files is a name associated with a specific point in the program's AS, such as a function or a global variable. Once the final position of the object file is known during linking, the correct addresses of the symbols can be entered at the appropriate places. As shifting a portion of the memory image is essentially a relocation of a part of the program, the relocation entries can be used in much the same way to fix addresses to symbols whose position changed by the insertion.

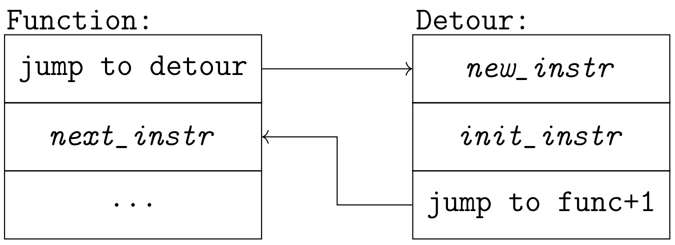
However, while the compiler builds these relocation tables when generating object files, they are typically consumed by the linker and not rebuilt when the final executable/shared object is created. For this reason, the relocation tables are unavailable in the files on which CGPoPE operates, and can't be used.

(2) Position Independent Code. PIC is machine code that can be properly executed regardless of its absolute address. This can be achieved by redirecting function calls through a procedure linkage table (PLT) and data references through a global offset table (GOT). PLTs contain stub functions that resolve the actual address of the function at run time. Likewise, GOTs contain the actual run time addresses of variables. When shifting parts of the memory image the tables can be adapted, so that the references are correctly resolved at run time.

But in addition to GOTs and PLTs, position independence can also be achieved by program counter (PC)-relative instructions. When an instruction contains a PC-relative address, it references a position in the AS relative to



(a) The original function



(b) The hijacked function

Figure 3.3: The function hijacking mechanism

the current state of the PC, i.e. its own position in the AS. If all references to a symbol are made this way, there is no need for it to have an entry in the GOT/PLT, as long as the same offset applied to both the instructions and the referenced symbol. This requirement is satisfied when mapping a program's memory image at an arbitrary offset, but not when shifting only parts of the program, because for this, the offset is only applied to the part of the memory image after the inserted instruction. For this reason PIC is unsuitable for CGPoPE when PC-relative addressing is used. Most architectures (including x86) have a PC-relative addressing mode, and compilers take advantage of this when generating PIC.

(3) Function Hijacking. The third approach bypasses the need to shift parts of the program entirely. It is based on the observation that while one cannot simply *insert* instructions, one can append new instructions at the end of the memory image and *replace* instructions anywhere in the binary. The execution flow of a function can then be *hijacked* as shown in Figure 3.3. We consider a function that starts with the instructions *init_instr* and *next_instr* (Figure 3.3a). Figure 3.3b shows the function after applying the mechanism as follows. First, CGPoPE allocates sufficient space at the end of the memory image, and fills it with the new instruction *new_instr*. We call this space the *detour* of the function. For the new instruction to be executed, the initial instruction *init_instr* of the function is replaced with a jump to the detour, labelled '`jump to detour`' in the figure. The replaced instruction needs however be run, as to not interfere with the functions functioning. They are copied to the detour, so that they are executed when it is taken. The last instruction in the detour is a jump back into the function, more precisely, to the next instruction *`next_instr`* after the jump to the detour. This operation is designated as '`jump to func+1`' in the figure. Thus, all the original instructions in the function are executed, in addition to the new instruction, and normal execution of the function can resume.

As the first two approaches, the usage of either relocation tables or PIC, proved to be inapplicable for instruction insertion on most binaries, this third approach, function hijacking, is used by CGPoPE. We now discuss additional problems that arise when applying function hijacking to precompiled executable.

The aforementioned PC-relative addressing mode induces a problem in the function hijacking mechanism. If the initial instruction of a function contains a PC-relative reference, moving it to the detour would break its functioning, as its position relative to the remainder of the program would be changed. The mechanism thus cannot be applied to functions beginning with such instructions.

As position independence by PC-relative addressing comes with no additional costs, many programs are compiled position independent. It is therefore desired that CGPoPE can operate on such binaries, without discarding the position independence. By using PC-relative addressing for all new instructions added, i.e. the jumps and the injected instructions, CGPoPE produces only PIC, thereby retaining any position independence of the augmented binary.

3.3.2 Prefetch Code Injection

With the mechanism presented in Section 3.3.1 we can now inject instructions into functions. From Section 3.2.3 we know which of its callees a function should prefetch. To combine these two results, we need method to issue prefetches.

Usually, the only way to direct the hardware to fetch specific data (which can include code, i.e. instructions) into the cache is by accessing it using a load-/store operation. The CPU then likely also copies the read/written data into the cache. The policy by which existing entries are evicted and the cache is managed is in this case solely hardware-defined. Fetching instructions into the cache by accessing their memory location is unsuitable for prefetching: Even though, on an out-of-order CPU, this "prefetch-by-access" could be executed without stalling subsequent instructions, explicit prefetch support from the hardware is still preferable, as it requires less logic to be executed by the CPU than a complete memory access, for example, the data only needs to be fetched into the cache, and does not have to block a register. Explicit prefetching also enables the application to control into which cache level the instruction is prefetched. which is especially of interest when data and instruction caches are seperated. If the instruction cache is seperate on at least one level (e.g. on Intel instruction and data caches are seperate on level 1), then prefetching an instruction by accessing its memory location would pollute the data cache.

Thus, the process of prefetching data/instructions must be specially supported by the hardware. More precisely, a machine instruction suitable for fetching the callee function into the cache must be provided by the instruction set architecture (ISA). Most CPU vendors provide such a *prefetch instruction*. For example, Intel provides one via the x86 Software SIMD Extensions (SSE) [16] and AMD via 3DNow! [1] and SSE [2].

With this, we can insert prefetch instructions at the appropriate places into the binary. CGPoPE iterates over all edges of the filtered call graph. For each of these edges ⟨*caller, callee*⟩, CGPoPE inserts a prefetch instruction for *callee* at the beginning of *caller* using the function hijacking mechanism. Thereby, each function prefetches its most frequent callee. The detours of all functions are placed consecutively at the end of the memory image of the program, after any text or data sections. This has the advantages of allowing a simple incremental allocation scheme for the detours when processing the binary.

Chapter 4

Implementation

In this work, we present *Call Graph Prefetching on Precompiled Executables (CGPoPE)*, a system for injecting instruction prefetches into application binaries based on the call graph of the application. In this chapter, we present our implementation of CGPoPE, which operates on application binaries in the executable and linkable format (ELF) [38] and x86 [8] machine code.

Chapter 3 presented the general design of CGPoPE, whereas this chapter focuses on issues specific to the x86 architecture and ELF.

The CGPoPE implementation primarily consists of a C program and a SystemTap [36] script. Section 4.1 gives a brief overview of the components and their interaction. Section 4.2 discusses the collection of profiling data, which is the part of CGPoPE not implemented in C. Finally, Section 4.3 covers the main program of CGPoPE, which performs the post processing of the profiling data and the actual augmentation of the application binary.

4.1 Overview

Our implementation of CGPoPE is structured into two main components: a SystemTap script and the main program. The SystemTap script monitors the application and reports function call events to the main program, which then processes the events and augments the binary of the application with prefetch instructions based on these calls.

An overview of this process is given in Figure 4.1. On the left side, the figure shows the SystemTap script and various files CGPoPE interacts with, and on the right side, a simplified view of the main program. Its four modules, together with the SystemTap script comprise the core logic of CGPoPE. The main module `augment` serves as a controlling instance, calling the other modules and passing data between them. The other modules do not

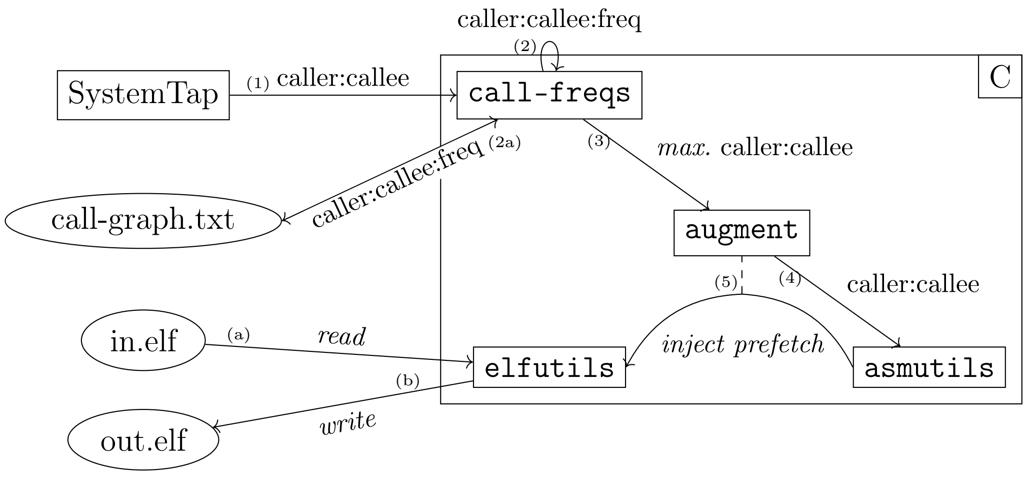


Figure 4.1: Overview of CGPoPE implementation

directly interact with each other and each encapsulate functionality of the core CGPoPE components: `call-freqs` manages and stores the call graph, `elfutils` parses, modifies, and writes ELF files, and `asmutils` comprises x86 machine code handling, like instruction en-/decoding and alteration. The primary workflow, as depicted in Figure 4.1, is as follows. First, the application given as *in.elf* is started and monitored by the SystemTap script (profiling run). (1) The script emits *caller:callee* tuples, which are then read by the `call-freqs` module. (2) The `call-freqs` module accumulates multiple occurrences of the same tuple to a *caller:callee:frequency* tuple and (2a) optionally writes this *preprocessed* data to a persistent file, so that they can be later reused for augmentation. Therefore, CGPoPE can either operate on unmodified input from step one, or load data already preprocessed by itself from disk. This is for example useful to perform multiple augmentations (possibly with different optimisation parameters) without reprofiling the application, as profiling might take a long time. (3) After accumulating the 2-tuples into 3-tuples, or loading them from disk, they are filtered to contain only the most frequent callee for each caller. (a) The application binary *in.elf* is then read and parsed by the `elfutils` module. (4) To insert the prefetches, the `augment` modules passes each of the remaining *caller:callee* pairs to the `asmutils` module, to encode and (5) inject a prefetch instruction for the callee into the caller. The injection of the encoded instructions is performed by the `elfutils` module, which (b) writes the augmented binary to the output file *out.elf* in the end.

```
#!/bin/stap
probe process.function("*").call {
  printf("%s:%s\n", stack[call_depth-1], ppfunc())
  stack[call_depth++] = ppfunc()
}

probe process.function("*").return {
  call_depth--
}
```

Listing 4.1: Simplified version of CGPoPE's SystemTap script

4.2 Call Interception

As presented in Section 3.2.2, part of CGPoPE is to trace function calls, in order to construct the call graph from these call records. In this section, we discuss two approaches for implementing function call tracing. We demonstrate why the straightforward approach of using perf's stack tracing [24] is unsuitable and what the advantages of using SystemTap [36] are.

perf. perf [24] is a user space tool and kernel subsystem for Linux to use hardware performance counters and trace kernel events in applications, such as system calls and scheduling events. Using perf to trace function calls would however require the usage of a function call event counter, which is not available on most CPUs (see Section 3.2.2). Another option to make use of perf would be to use its built-in call graph feature. Perf's method is to include snapshots of the current stack in each sample, which are taken either periodically or when a monitored event counter surpasses a given threshold. The stack traces can then be accumulated into a call tree. Although this method is suitable to identify "hot" subtrees in the program's call graph, it gives no notion of how often which call was actually made: If two stack traces both contain the call *caller*→*callee*, this can either be because *caller* called *callee* twice, or because so much time was spent in that subtree that two samples were taken in that time.

SystemTap. We instead require a method that allows us to intercept and record individual function calls. SystemTap [36] is a user space tool and scripting language for Linux to dynamically instrument running kernels and user space processes. It utilises the kernel's *Kprobes* and *Uprobes* APIs to execute arbitrary code (*probes*) at certain predefined events. Whereas

SystemTap can also place probes in kernel code, especially the probing of user space processes is of interest for our purpose. There is a probe point `process .function(PATTERN).call` to intercept calls of functions matching `PATTERN`. This probe point is used by CGPoPE to record function calls. A simplified version of the script is shown in Listing 4.1. In the `.call` probe, the function call is printed as *caller:callee*, the caller beeing read from the stack. Then the current callee is pushed to the stack as the new caller. In the `.return` probe, the topmost element of the stack is discarded, to restore the previous caller for all subsequent calls.

To probe user space events SystemTap uses the Uprobes API [19], which instruments the code of the monitored application to trap into the kernel when the probe point is reached, in our case call and return instructions. Whereas this allows to intercept each individual call instruction, which enables us to later build an exact and complete call graph, it also comes at the disadvantage of having to trap into the kernel at every function call, which is very resource-intensive.

Because the amount of data might be very large when recording every individual function call, the accumulation of identical records, i.e. calls that occurred more then once, into single records is performed live, as described in Section 3.2.2. The SystemTap scripting language has support for associative arrays, so one might use a statement like `++freqs[caller,callee]` to perform the accumulation within the script. However, although the accumulated records are considerably smaller than that of the individual calls, their size is still in $\mathcal{O}(n^2)$, n being the total number of functions. This is still be too large for reasonably complex applications. SystemTap limits the size of associative arrays to 2048 entries by default for safety reasons [37], because the associate arrays are allocated in kernel memory.

To avoid storing a large amount of data in kernel memory, we perform the accumulation in user mode. The SystemTap script thus only prints the *caller:callee* records, whereas the accumulation is handled by the main program.

4.3 The Binary Augmenter

Now, the function call records produced by the SystemTap script need to be accumulated into a call graph as described in Section 3.2.2. Based on the generated call graph, prefetches are injected into the application binary. The accumulation and injection are handled by the main program of our CGPoPE implementation, the *binary augmenter*, which we present in the following section. Section 4.3.1 describes how the binary augmenter

```
function AugmentBinary(input_elf, call_records)
    extract prefetch_targets from call_records          ▷ Section 4.3.1
    setup input_elf                                     ▷ Section 4.3.2.1
    for caller,callee in prefetch_targets do
        inject <prefetch to callee> into caller         ▷ Section 4.3.2.2
    return output_elf
```

Listing 4.2: CGPoPE Main Loop

handles the accumulation of call records the SystemTap script has produced, and Section 4.3.2 describes the implementation of the *function hijacking* mechanism introduced in Section 3.3.1. Listing 4.2 gives an overview of the performed tasks. Whereas the procedures described in Sections 4.3.1 and 4.3.2.1 are performed only a single time for the whole binary, the instruction injection described in Section 4.3.2.2 is performed once for each prefetch that is to be injected.

4.3.1 Call Accumulation

The `call-freqs` module handles the creation of the call graph. The module generates the call graph in its hash table, and optionally its 3-tuple representation, as defined in Section 3.2.2. The module also performs the selection of a suitable prefetch target for each function. As described in Section 3.2.3, we use the most frequent callee of each function as the prefetch target.

The `call-freqs` module reads the records from a file stream. This stream can either be directly attached to the output of the SystemTap script, or to a file containing the saved 3-tuple representation of the call graph, i.e. the output of this step, the `call-freqs` module. In both cases, each line of input comprises a record. In the first case, each record represents a single call event, in the second case, each record states how often a call event occurred. The parser is implemented in such a way that it can read both types of records compositely, so that the input can be read and interpreted independently of its source. The exact record format complies with the form *caller:callee:n*, which reads as "*caller* called *callee* n times". Alternatively, the format is *caller:callee*, in which case $n = 1$ is assumed.

To build the hash table representation of the call graph, we use the UThash [13] library. The `call-freqs` module iterates over each input record and looks up the entry with the key *caller:callee* in the *call graph table*. If the entry exists, its value is incremented by n, otherwise it is created and initialised with n.

Once all input records are processed and the hash table representation of the call graph is built, its 3-tuple representation can optionally be dumped to an output file. This file can later be re-read using the procedure described in this section.

The `call-freqs` module determines the most frequent callee for each function from the resulting call graph. It uses a second hash table, the *prefetch target table*, to store the most frequent callees, mapping each caller to its most frequent callee. To perform the extraction of the most frequent callees from the call graph, it first sorts the entries in the call graph table so that all entries for a caller are adjacent and the most frequent callee for each caller appears first. Such an order can be achieved by sorting the entries alphabetically by the caller name, and by the frequency n within each caller. Then, the `call-freqs` module iterates over the entries in this order and extracts the most frequent callees by transfering only the first entry for each caller to the prefetch target table, which is the one with the maximum n for this caller. The adjacency of all entries for the same caller eases the check if the currently processed entry is the first for its caller: When all entries with the same caller occur consecutively, an entry is the first one for its caller, if and only if the previous entry had a different caller.

4.3.2 Function Hijacking

We now have selected a set of suitable prefetch operations, in the form of $\langle$*caller,callee*$\rangle$ pairs stored in the prefetch target table.

To insert the prefetch instructions into the application binary, we implement the *function hijacking* mechanism introduced in Section 3.3.1. Function hijacking is a mechanism to inject arbitrary instruction at the beginning of a function, which can be used to inject prefetch instructions. The mechanism places the to-be-injected instructions in a special *detour* at the end of the application's memory image and replaces the initial instructions of the function with a jump instruction to that detour. A jump back to the function is placed at the end of the detour.

In this section we present our implementation of function hijacking. Building upon the general mechanism formulated in Section 3.3, this implementation applies function hijacking to ELF binaries containing x86 machine code.

4.3.2.1 ELF File Modification

For the implementation of function hijacking, sufficient space to hold the detours for all functions is required. Because shifting instructions and thus the insertion of code in the middle of the address space (AS) is impossible, we place the detours at the end of the AS. The `elfutils` module performs the setup of the input binary and the allocation of sufficient space for the detours.

ELF files are structured in multiple *sections*, each containing various information relevant to the application. Sections can comprise data or executable code to be mapped into the main memory upon execution [38], or control structures such as relocation or symbol tables that are not mapped into the application's AS. Sections that are to be mapped are marked as *allocated* with the `SHF_ALLOC` flag. All allocated sections usually appear consecutively, so that they can be mapped into the main memory with a single operation. The end of the mapped part of the ELF file can thus easily determined by identifying the last allocated section in the file. Any following sections are not mapped at run time and can thus easily be shifted. The `elfutils` module can then insert a new executable *detour section* of sufficient size after the last allocated section to extend the application's memory image.

The operating system needs to be advised to load the new section when the application is started. As the section header table offers a very detailed view of the ELF file, which is not required by the operating system to load the memory image, the mapping of sections to the main memory is specified in a separate *program header*. Each entry in this table designates a *segment*, which specifies a portion of the file (offset and size), a corresponding region in the AS (address and size) and a set of access flags (writable, readable, executable). An ELF file usually contains at least two segments: a code segment, which contains executable components of the application, such as the text section; and a data segment, which primarily includes the data and bss section. The most straightforward approach would be adding a new segment to the program header. This is, however, problematic, because the program header is located at the beginning of the file and often mapped into the AS, so extending it would again require to move subsequent (allocated) sections, which is not possible. Although the program header could theoretically be moved to the end of the memory image, for reasons of simplicity the `elfutils` module instead increases the size of the preceding segment to include the detour section and ensures it is marked as executable. Note that this comes at the disadvantage of making the whole data segment executable, which is considered a security concern, as it potentially enables an attacker to divert the control-flow to previously placed malicious code [34]. This is usually prevented by ensuring that no memory segment is both writable and executable [40], an assumption which no longer holds when making the data segment executable.

Special care needs to be taken if the original last allocated section is the bss section, i.e. marked as type `SHT_NOBITS` in the file. The bss section contains application data that is to be initialised to zero. In ELF this is achieved by placing the bss section last in the segment, and setting the in-memory size of the segment larger than the in-file size of the segment, the difference being exactly the size of the bss segment. This way, when the operating system loads the contents of the ELF file into the main memory, the "missing" part of the segment, which corresponds exactly to the location of the bss section, is initialised by the operating system with zeros. This mechanism however fails, when the bss section is not the last section of the segment, which is the case after appending the detour section. As a simple workaround, the `elfutils` module converts any preceding section of type `SHT_NOBITS`, if one exists, to a regular section (type `SHT_PROGBITS`) filled with zeros in the file and adapts the program header accordingly. The disadvantage of converting the bss section is that it unnecessarily increases the size of the executable. The conversion introduces a portion in the file that is merely filled with zeros and thus contains no information which could not also be expressed by marking the section as `SHT_NOBITS`. The need to convert the bss section could be avoided by placing the new detour section in a separate segment, which would, however, require the additional effort of extending the program header.

4.3.2.2 x86 Instruction Injection

At this point, we have allocated sufficient space in the application's binary to store the detour of each function, and can thus apply the hijacking mechanism to each function, which is implemented in the `asmutils` module.

As stated in Section 3.3.2, we require a special instruction to perform the prefetch. Software SIMD Extensions (SSE) provides instructions for x86 to fetch data into caches [16]. Whereas there is no instruction to explicitly prefetch instructions, since instruction and data caches are only separated on level 1, the same instructions used to prefetch data can be used to prefetch instructions from the level 2 cache (L2) upwards. We use the `PREFETCHT1` instruction, which prefetches into the L2. Note that the unavailability of a suitable level 1 instruction cache (L1-I) prefetch instruction limits CGPoPE on x86 to the mitigation of L2 misses, even though, as noted in Section 2.2, L1-I misses are also problematic.

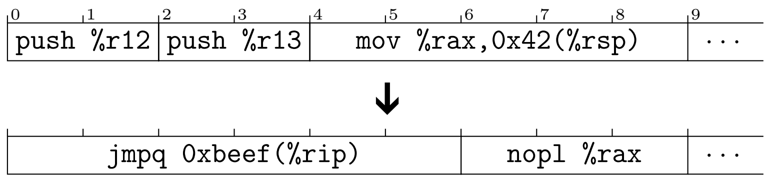


Figure 4.2: Example function with x86 instructions of varying length, before (top) and after (bottom) applying function hijacking

To implement function hijacking, we need to perform various instruction level operations. These operations include encoding the to-be-inserted prefetch and the jump instructions, as well as decoding the instructions at the beginning of a function to verify that they can be moved. To encode and decode instructions, the `asmutils` module uses the X86 Encoder Decoder (XED) library [7].

A special property of x86 are its varying instruction lengths: Encoded instruction can be 1 to 15 B long. To find out the length of the instruction at a given position, it has to be partly decoded, a feature which is provided by XED.

The arbitrary instruction length can result in a problem when a function is hijacked, more precisely when the initial instruction is replaced with a jump to the detour: Relative jump instructions, which we use for function hijacking (see Section 3.3.2), can be up to six bytes long, while the initial instructions of many functions are typically stack management operations such as small two-byte push instructions. In this case replacing the first instruction of a function does not yield sufficient space to hold a jump, and the `asmutils` module has to move multiple instructions to the detour. Consider the example in Figure 4.2. It shows an example function that starts with two two-byte push instructions and a five-byte load instruction. In this example the jump to the detour is a instruction with a length of six bytes. To fit it at the beginning of the function, we have to move at least the two push instructions to the detour. Since this still leaves us two bytes short, and instructions cannot be split, we have to move the load instruction to the detour, too. We then place the jump instruction and fill the superfluous space with a three-byte NOP instruction. The jump back from the detour to the function skips execution of the NOP, by targeting the instruction immediately after it, which in the example is the ninth byte of the function.

Another issue is that applications can contain very small functions, usually generated by compilers, sometimes only a few bytes long. If a function is smaller than the length of the required jump instruction, the function hijacking mechanism cannot be applied and it is skipped by CGPoPE. Since such functions are relatively rare, for example recent builds of the database severs Redis [29] and PostgreSQL [27] contain none, we do not expect this to have any considerable impact.

Chapter 5

Evaluation

In this work we present *Call Graph Prefetching on Precompiled Executables (CGPoPE)*, a system to mitigate high instruction cache miss rates by injecting prefetch instructions into application binaries. As described in Chapter 3, CGPoPE achieves this injection by hijacking the execution flow of each function and redirecting it to a special code section which contains the prefetch logic, before resuming normal execution of the function. The hijacking is performed by replacing the initial instructions of functions with jump instructions.

In this chapter we present our evaluation of CGPoPE. We first introduce our evaluation environment and methodology in Sections 5.1 and 5.2. In Section 5.3 we present our results: Although we find CGPoPE unable to improve the performance of the benchmarks or to reduce overall instruction cache miss rates, we show that CGPoPE does issue effective prefetches for the most frequent callees of each function. Finally, we analyse our results in Section 5.4, putting them in context with our design and implementation, and discuss directions to improve CGPoPE.

5.1 Evaluation Environment

We conduct our evaluation on an Intel Core i5-6500 CPU, comprising 4 physical cores with a frequency of 3.20 GHz each. Each core has an independent level 2 cache (L2) with a capacity of 256 KiB. A L2 next-N-line (NL) instruction prefetcher is active for each CPU on the system [41]. No level 1 instruction cache (L1-I) prefetchers have been disclosed to be implemented by Intel. All benchmarks are run on a single system, so the communication between the server and the client side of the database benchmarks does not involve network communication. The system is running Ubuntu 17.04 with a Linux 4.11.3 kernel.

5.2 Methodology

The augmentation of the benchmark binaries is performed using the CGPoPE implementation presented in Chapter 4. The runs of the unmodified and augmented versions of the benchmarks are then monitored and their performance is measured as described in Section 5.2.1. In Section 5.2.2 we present the benchmarks we use.

5.2.1 Measurements

Because no L1-I prefetch instruction is available on x86 computers, the implementation only injects prefetches for the L2. Therefore, we only consider L2 misses as a suitable metric to measure the effectivity of CGPoPE.

To allow a more detailed analysis of the cache behaviour induced by CGPoPE into the benchmarks, we distinguish between three types of cache misses, based on the target instruction that was missed:

- If the missed instruction was the first instruction of a function, we call the miss a *call miss*. It indicates a miss that was directly caused by a function call.

- *Detour misses* are misses of additional instructions introduced by CGPoPE. CGPoPE hijacks the execution flow of each function and redirects it to a *detour*, which contains the prefetch instructions. The detours are placed in a special section in the application's memory. Since executing the detours involves jumping to them, they are a likely cause of additional instruction cache overhead. Detour misses are easily identified by their target addresses being located in the detour section, which is outside of the regular text section of the application.

- All other misses are *local misses*, i.e. misses caused within function boundaries by sequential execution or low-distance control-flow branches.

To measure L2 misses, we use the `FrontEnd_Retired.L2_MISS` performance counter provided by Intel's Process Event Based Sampling (PEBS) facility [16]. PEBS events are *precise* events, meaning that the exact instruction that caused the event can be determined. Only a precise event is suitable to distinguish the three miss types. To collect performance counter data we use the Linux perf tool [24].

We also assess the *effectivity* of individual prefetch operations. A prefetch is said to be effective, if and only if it is *timely* and the data is still present in the cache once it is accessed. A prefetch is timely if and only if the data brought into the cache is accessed only after the prefetch request is completed [25]. For instruction prefetches, this especially means that the prefetched function is actually called before it is evicted from the cache.

5.2.2 Benchmarks

We conduct our evaluation using three benchmarks. The TPC-C and the *redis-benchmark* were chosen to represent realistic database workloads. The *functree* benchmark is a microbenchmark developed by us to allow a detailed analysis of the cache behaviour induced by CGPoPE.

redis-benchmark. The `redis-benchmark` tool is the builtin benchmarking utility shipped with the Redis in-memory NoSQL database [29]. Although the `redis-benchmark` is no traditional SQL online transaction processing (OLTP) workload, we include it in our evaluation to represent the typical workload of a modern NoSQL key-value stores. The benchmark issues a number of different operations to the Redis server concurrently, using multiple parallel network connections. Note that the server itself handles all request in a single thread. We configure the `redis-benchmark` to issue 100 000 requests via 50 parallel connections.

TPC-C. The TPC-C benchmark is a common OLTP benchmark to measure and compare the performance of database management systems (DBMSs). It has been shown to be one of the most instruction cache-bound benchmarks, with L2 miss rates of up to 75 misses per kilo-instruction [10]. TPC-C is therefore a suitable choice to assess the quality of CGPoPE optimisations for DBMSs. We use the OLTP-Bench [9] implementation of TPC-C and run it against a PostgreSQL DBMS server [27]. Since, in contrast to Redis, PostgreSQL handles requests in parallel by forking multiple processes, we consider the TPC-C benchmark in two different variants: *TPC-C/S* disables parallelism by binding all server processes to a single CPU using the Linux utility `taskset`, while *TPC-C/M* allows the server processes to be executed on different CPUs. We configure the TPC-C load with four warehouses and four clients.

```c
void leaf_aa() { asm ("nop; [...]"); }
// [...] 'leaf_a(b|c)' ommitted
void node_a() {
  asm ("nop;nop;[...]"); // times n
  for (int j=0; j<10; ++j) leaf_aa();
  leaf_ab(); leaf_ac();
}
// [...] 'node_b' and 'leaf_b(a|b|c)' ommited
void root() {
  asm ("nop;nop;[...]"); // times n
  node_a();
  for (int j=0; j<10; ++j) node_b();
  node_c();
}
int main(int argc, char **argv) {
  __clear_cache(0, (void*) -1); // 0x0 to 0xf..fff
  for (int j=0; j<(atoi(argv[1])); ++j) root();
  return 0;
}
```

Listing 5.1: Schematic view of a source code file generated by `functree`

functree. We developed `functree` as a tool to generate microbenchmarks with high instruction cache overhead. We perform the generation by automatically producing C source code based on a set of parameters and compiling it with GCC [11]. `functree` generates a w-ary full function tree of depth d. Listing 5.1 shows a schematic view of a source code file generated by `functree` generated with $w = 3$ and $d = 2$. Each node function calls each of its w children a single time, except one child, which is called multiple times (10 times in the example) so that it stands out as the most frequent callee. In the example, `leaf_aa` is the most frequent callee of `node_a`, and `node_b` of `root`. To ensure that the instruction working set size exceeds the L2 capacity, we increase the size of each function sufficiently, by filling each function with n bytes of NOP instructions. The NOPs are generated using the C inline assembler directive at the beginning of all tree functions. To ensure that results stay consistent across multiple runs we use the GCC builtin function `__clear_cache` [12] to flush the instruction cache at the beginning of each run. Since from the L2 upwards data and instruction caches are shared, we also execute the function tree i times, thereby creating identical cache states for all iterations but the first, and thus further ensuring homogeneity of the results. As seen in the figure, i is read from the command line, so that it can be changed without regenerating the benchmark.

We found a six-ary tree of depth six with 2048 bytes of NOPs per function executed 10 times to be suitable to emulate instruction cache bound execution ($w = d = 6, n = 2048, i = 10$).

5.3 Results

To measure the effect of CGPoPE on the benchmarks' performance, we measure the absolute L2 instruction misses for an unoptimised and an optimised version of each benchmark. The optimised versions are generated by applying CGPoPE to the unmodified server binary used in each benchmark. In Section 5.3.1 we find that CGPoPE does not reduce the overall instruction cache miss rates of the OLTP benchmarks, and in Section 5.3.2 we investigate the cause for CGPoPE's ineffectivity. All results presented in the following sections are obtained by running each of the experiments ten times and calculating the arithmetic mean of each value.

5.3.1 Real-World Applications

We first examine the effect of CGPoPE on the overall L2 instruction miss rates of TPC-C (single- and multicore) and `redis-benchmark`. These are depicted in Figure 5.1. The figure shows the absolute number of L2 instruction misses for each of the benchmarks, when run with a unmodified (blue) and a augmented binary (red). While we note that `redis-benchmark` has a significantly lower L2 instruction miss overhead than the traditional OLTP workload of TPC-C, we find that for all benchmarks the difference in the miss rates of unmodified and augmented binaries are below 5 % and have no notable effect on the benchmarks' performance: For TPC-C/S and TPC-C/M, the performance varies between 378 and 438 requests per second, and the `redis-benchmark` takes approximately 17 seconds to complete, both independently from whether CGPoPE is in effect.

We also observe that the instruction cache miss rate of TPC-C/M is 8.91 % lower than that of TPC-C/S. A possible reason for this is that, although all PostgreSQL server processes are running the same binary, they still have sufficiently different instruction working sets to benefit from having distinct L2 instruction caches. Because the parallel TPC-C/M uses multiple CPUs, it is more strongly affected by scheduling effects, since the server processes are now subject to CPU migrations and the chance for them being interrupted by other tasks is higher than when they share a single core.

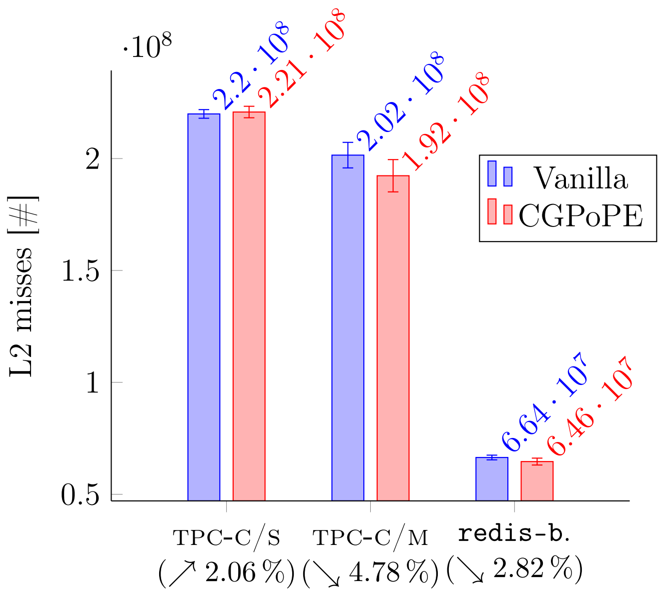


Figure 5.1: Instruction cache misses of unoptimised and optimised benchmarks

Therefore, the performance of TPC-C/M shows increased non-determinism in comparison with TPC-C/S, which is reflected by the higher standard deviation of 2.8 to 3.7 %.

We conclude that CGPoPE does not reduce the L2 instruction miss rates significantly and has no measurable effect on the benchmarks' performance.

5.3.2 Synthetic Benchmarks

We use the `functree` microbenchmark to investigate possible reasons for the ineffectivity of CGPoPE and to identify misses that are effectively prefetched. Figure 5.2 shows the results of comparing the L2 misses of an unoptimised and an augmented binary. The figure shows the absolute L2 instruction misses measured in samples (each sample corresponds to 100 007 actual misses) overall, as well as for local, function and detour misses specifically. We observe that the total miss rate is slightly reduced by 5.3 %. This reduction is not enough to significantly improve the performance, the run time stays constant at 176.34 s. Thereby, these results are consistent with those for the database benchmarks presented in Section 5.3.1. To investigate how the reduction was caused we look at the three miss types separately: The majority of instruction cache misses in `functree` is caused by local misses, which are not significantly decreased by CGPoPE, since they are not target

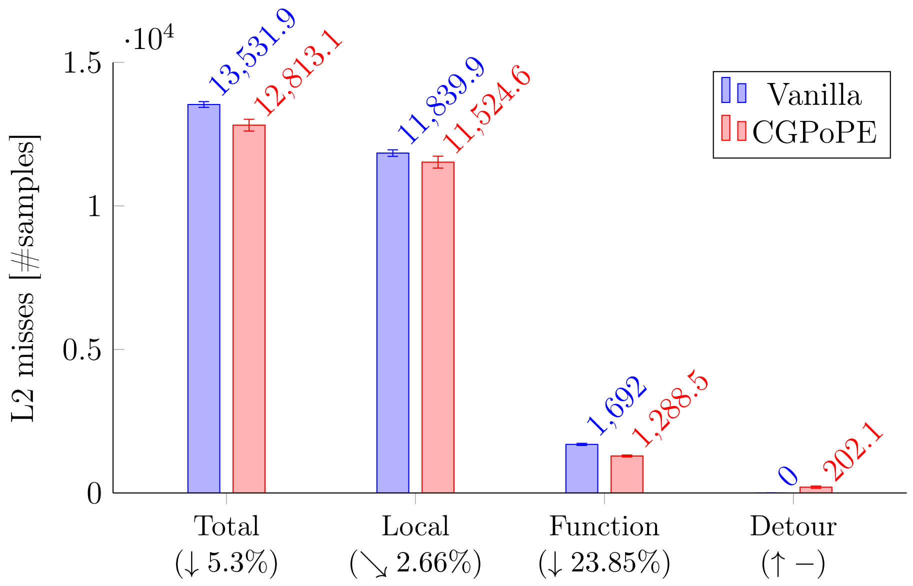


Figure 5.2: Instruction cache misses of `functree` by type

of our optimisation. Only about 12.74 % of all L2 misses are function misses. Regarding the function misses in separate, we find that applying CGPoPE reduces the misses by 23.85 %, which is significant. When CGPoPE is applied to a `functree` binary, it injects a single prefetch per function (to the most frequent callee). Each function has $w = 6$ children in our configuration, leading CGPoPE to issue prefetches for $\frac{1}{6} = 16.\bar{6}\,\%$ of all functions. This number roughly corresponds to the 23 % reduction of function misses, and therefore the prefetches issued by CGPoPE are actually effective. The cache miss reduction is even larger then $16.\bar{6}\,\%$. The cause for this can be found in the iterative calling of subtrees in `functree`: Every function calls each of its children only a single time, except one of its children multiple times, in the form of a `for`-loop, so that a single child of each function stands out as the most frequent callee, and is then selected as a prefetch target. For this reason, even though the fraction of functions that are prefetched is exactly $\frac{1}{6}$, the fraction of calls that are made to those functions is actually higher, because they are called most often.

The analysis also reveals that CGPoPE introduces detour misses as a new potential cause of performance overhead. Detour misses, however, only account for about 0.92 % of all L2 instruction misses, and are therefore an unlikely cause of significant performance overhead.

To show that the detours do not significantly impact the performance we measure their exact CPU overhead, by applying a slightly adapted variant

	Vanilla	NOP-CGPoPE	Relative overhead
Cycles	$6.358 \cdot 10^{11}$	$6.363 \cdot 10^{11}$	0.07 %
Instructions	$2.5107 \cdot 10^{12}$	$2.511 \cdot 10^{12}$	$9.72 \cdot 10^{-3}$ %
Run time	176.45 s	176.42 s	−0.02 %

Table 5.1: Processor overhead induced by detours

of CGPoPE to `functree`: Instead of prefetch instructions, we inject NOP instructions into the function. This way, we only introduce the potential overhead caused by the detours, without any positive effect via prefetches, and can measure the detour overhead. Table 5.1 shows the total number of cycles, instructions, and the run time of unmodified and NOP-augmented `functree` runs. As CGPoPE only adds three additional instructions per function (jump to the detour, prefetch and jump back to function), opposed to the approximately 2070 instructions per function in `functree` (2048 NOPs plus children function call logic), the relative increase in instructions executed is close to zero, namely below 0.01 %. We find the total overhead caused by CGPoPE to be insignificant, with an increase in required CPU cycles of only 0.07 %. The total run time was even decreased by 0.02 % in our experiments, which is likely to be attributed to unrelated hardware or scheduling effects.

5.4 Analysis

Although we found CGPoPE unable to reduce the overall L2 instruction miss rates, we also showed that the prefetches issued by CGPoPE are effective most of the time. Possible reasons for not all prefetches being effective include that the most frequent callee might be called at a relative late point in the function. As CGPoPE injects all prefetches at the beginning of functions, the prefetched cache line might already be evicted once the call is performed. To increase the effectivity of prefetches of most frequent callees that are not the first call in the function, an approach might be to move the prefetch closer to the corresponding call instruction, for example shortly after the previous call, similar to the original Call Graph Prefetching (CGP) software implementation by Annavaram, Patel and Davidson Annavaram, Patel and Davidson. This would ensure that no potentially very deep function tree is executed inbetween the prefetch and its corresponding call.

We also observed that the majority of instruction misses in `functree` are local misses, whereas CGPoPE is only able to mitigate function misses. This observation seems to contradict with the statement found in literature and described in Chapters 1 and 2, that the majority of instruction cache misses in scale-out and OLTP workloads is caused by long-distance branches like function calls, and execution within functions is captured by the NL prefetcher. The NL prefetcher does, however, only prefetch the next N lines relative to the current program counter (PC). Therefore, when prefetching a target function, only the cache line containing the first instruction of the function is prefetched. The first cache line comprises, depending on the alignment of the function, maximally the first 64 (the cache line size in Intel's L2 cache) bytes of the function. Instruction cache misses caused by long-distance branches thus do not only include the first instruction of the function, but also all subsequent instructions, because the NL prefetcher only starts to prefetch the instructions of the function after the jump is performed. Because CGPoPE only prefetches the first line of the target function, the subsequent instructions of the function are left to be missed upon entering the function. The prefetching of the first N lines of a function could be achieved in hardware or in software: If the CPU were aware that the prefetch target is an instruction, for example via a special instruction prefetch instruction, it could advise the NL prefetcher to start prefetching relative to the prefetched target. As it is unlikely that such a feature is available on production CPUs in the near future, CGPoPE could manually prefetch the first N lines of the most frequent callee by injecting N separate prefetch instructions for consecutive cache lines within the target function into each function.

In our evaluation of `functree`, we also showed that CGPoPE effectively prefetches approximately $\frac{1}{6}$ of all calls, because only one prefetch per function is performed. An attempt to increase this fraction could be made by prefetching more than the most frequent callee of each function. Because approximately 80 % of all functions have less than eight distinct callees [4], prefetching the eight most frequent callees might be more suitable. This improvement of CGPoPE could easily be combined with moving the prefetches closer to their corresponding callees, as proposed above, by injecting a prefetch for the most probable next callee after each call instruction, thereby fully emulating the original CGP policy implemented by Annavaram, Patel and Davidson [4].

We showed that the total performance overhead introduced by the detours is negligible. Reasons for this likely include the small size of the detours The detours produced when applying CGPoPE to our benchmarks have an average size of approximately 20 B, therefore about three detours can fit in a single cache line. Because the detours are stored consecutively in the memory image, it is thus likely that executing a detour causes neighbouring detour to be "prefetched" into the instruction cache. The small overhead caused by the detours could be further reduced by explicitly prefetching the detours: When prefetching a functions most frequent callee, an additional prefetch to the most frequent callee's detour could be added.

We conclude that CGPoPE, in its current form, cannot effectively reduce instruction cache miss rates, and does not improve the performance of the applications. At the same time, we find that the prefetches that are issued by CGPoPE are effective most of the time, though they prefetch to few lines. The most promising approach to make CGPoPE effective, is thus to increase the number of functions prefetched, by prefetching more callee functions, and to increase the number of lines prefetched, by prefetching the first N lines of each callee, instead of only the first.

Chapter 6

Conclusion

In this work we propose *Call Graph Prefetching on Precompiled Executables (CGPoPE)* as a system to mitigate instruction cache overhead in cloud applications and database management systems (DBMSs). The scale-out and online transaction processing (OLTP) workloads typically found in these applications suffer from high instruction cache miss rates [10]. The reasons for poor instruction cache utilisation have been identified as large instruction working sets and irregular, non-sequential instruction access patterns [18, 4]. Because the hardware prefetchers found in most modern CPUs are not able to capture these complex patterns, many more intelligent instruction prefetching schemes have been proposed. However, most of these solutions are either implemented in hardware, and therefore unavailable on most production systems, or in the compiler, thus require recompiling the application to optimise it. With CGPoPE we developed a system that is entirely implemented in software, requiring no hardware support besides a suitable prefetch instruction, and still functions without special support from the compiler.

CGPoPE issues one prefetch per function, to the most frequent callee of that function. We determine the most frequent callees via monitoring the function calls made by the program and building the call graph of the application from the traced function calls. CGPoPE evades the need for compiler support by operating on compiled executables, augmenting the application's binary with prefetches to the most frequent callee of each function at the beginning of this function. The major challenge in augmenting the binaries is the injection of the prefetch instructions into the unmodified machine code, since instructions cannot be inserted by shifting the subsequent instructions, as doing so would break references to existing code. To circumvent this problem, we developed *function hijacking*, a mechanism to hijack the execution flow of a function, by replacing the initial instructions of a function with a jump to a special *detour*, which contains the prefetch instruction.

By developing function hijacking, we showed that it is possible to inject prefetch instructions into precompiled executables. Although being designed to inject prefetch instructions at the beginning of functions, function hijacking can be used to inject arbitrary instructions at any point in a function.

We describe a prototype implementation of CGPoPE, targeting binaries in the executable and linkable format (ELF) [38] and x86 machine code [8]. To monitor the application and build its call graph, we use the dynamic instrumentation and monitoring tool SystemTap [36].

We evaluated our implementation using the TPC-C benchmark as a workload representative for OLTP applications, and the Redis benchmark to represent modern noSQL key-value stores. Although we find CGPoPE to be ineffective in significantly increasing the performance of applications, we nevertheless show that it issues effective prefetches for the most frequent callees. The major reasons for CGPoPE's ineffectivity are the fact that it only prefetches a single callee per function, and the first cache line of each function, so that in the event of a call the first instruction in the second line of the function is already a miss again.

6.1 Future Work

In order to make CGPoPE an effective measure to improve instruction cache performance, we suggest a number of improvements to mitigate the issues we observe in Chapter 5:

The outstanding problem is that CGPoPE only prefetches the first line of each function. This can be mitigated by prefetching not only the first, but the first N lines of a function, either by executing N prefetch instructions, or via a special prefetch instruction, that advises the next-N-line (NL) prefetcher to prefetch the lines following the prefetched address.

To increase the number of prefetched function calls, we suggest a prefetch target selection policy closer to Annavaram, Patel and Davidson's original policy [4]. Instead of prefetching only the most frequent callee, the eight most frequently called functions should be prefetched at appropriate places in the function.

Even though we found the total instruction cache overhead caused by jumping to the detours to be minimal, it could nontheless be further reduced by prefetching not only the start address of the most frequent callee, but also its detour, as the detour is immediately jumped to and executed upon entering the callee function.

Bibliography

[1] Advanced Micro Devices Inc. *3DNow! Technology Manual*. Version G. 2000. URL: http://support.amd.com/TechDocs/21928.pdf (visited on 08/24/2017).

[2] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions*. Version 3.20. 2017. URL: http://support.amd.com/TechDocs/26568.pdf (visited on 08/24/2017).

[3] Advanced Micro Devices Inc. *Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors*. Apr. 15, 2017. URL: https://developer.amd.com/resources/developer-guides-manuals (visited on 09/02/2017).

[4] Murali Annavaram, Jignesh M. Patel and Edward S. Davidson. "Call Graph Prefetching for Database Applications". In: *ACM Transactions on Computer Systems (TOCS)* 21.4 (2003), pp. 412–444. DOI: 10.1145/945506.945509.

[5] Phil Bernstein et al. "The Asilomar Report on Database Research". In: *SIGMOD Record* 27.4 (Dec. 1998), pp. 74–80. ISSN: 0163-5808. DOI: 10.1145/306101.306137.

[6] Carlos Carvalho. "The Gap Between Processor and Memory Speeds". In: *Proceedings of IEEE International Conference on Control and Automation*. 2002. URL: http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf (visited on 09/02/2017).

[7] Mark Charney. *Intel X86 Encoder Decoder Software Library*. Version 10.0-84. Intel Developer Zone. 2017. URL: https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library.

[8] Ulan Degenbaev. "Formal Specification of the x86 Instruction Set Architecture". PhD thesis. Universität des Saarlandes, 2012. URL: http://scidok.sulb.uni-saarland.de/volltexte/2012/4707.

[9] Djellel Eddine Difallah et al. “OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases”. In: *Proceedings of the VLDB Endowment* 7.4 (2013), pp. 277–288. DOI: 10.14778/2732240.2732246.

[10] Michael Ferdman et al. “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware”. In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM. 2012, pp. 37–48. DOI: 10.1145/2150976.2150982.

[11] GNU Project. *GNU Compiler Collection*. Version 6.3.0-12ubuntu2. 2017. URL: https://gcc.gnu.org (visited on 08/26/2017).

[12] GNU Project. *Internals of the GNU Compiler Collection*. Version 6.3.0. 2016. URL: https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gccint (visited on 08/26/2017).

[13] Troy D. Hanson. *UThash*. Version 1.9.8-248-g4d197f2. 2017. URL: http://troydhanson.github.io/uthash (visited on 09/02/2017).

[14] Stavros Harizopoulos and Anastassia Ailamaki. “Improving Instruction Cache Performance in OLTP”. In: *ACM Transactions on Database Systems (TODS)* 31.3 (2006), pp. 887–920. DOI: 10.1145/1166074.1166079.

[15] Wei-Chou Hsu and James E. Smith. “A Performance Study of Instruction Cache Prefetching Methods”. In: *IEEE Transactions on Computers* 47.5 (1998), pp. 497–508. DOI: 10.1109/12.677221.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*. 2017. URL: https://software.intel.com/en-us/articles/intel-sdm.

[17] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Springer Science & Business Media, 2006. ISBN: 9780387281322.

[18] Svilen Kanev et al. “Profiling a Warehouse-scale Computer”. In: *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2015, pp. 158–169. DOI: 10.1145/2749469.2750392.

[19] Jim Keniston et al. “Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps”. In: *Proceedings of the 2007 Linux Symposium*. 2007, pp. 215–224. URL: https://www.kernel.org/doc/ols/2007 (visited on 07/29/2017).

[20] Aasheesh Kolli, Ali Saidi and Thomas F. Wenisch. “RDIP Return-Address-Stack Directed Instruction Prefetching”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2013, pp. 260–271. DOI: 10.1145/2540708.2540731.

[21] Jaekyu Lee, Hyesoon Kim and Richard Vuduc. “When Prefetching Works, When It Doesn’t, and Why”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.1 (Mar. 2012), 2:1–2:29. ISSN: 1544-3566. DOI: 10.1145/2133382.2133384.

[22] Chi-Keung Luk and Todd C. Mowry. “Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors”. In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 31. Dallas, Texas, USA: IEEE Computer Society Press, 1998, pp. 182–194. ISBN: 1581130163.

[23] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer Science & Business Media, 2008. ISBN: 3540779779.

[24] Arnaldo Carvalho de Melo. “The New Linux ‘perf’ Tools”. In: *Slides from Linux Kongress*. Vol. 18. 2010. URL: http://www.linux-kongress.org/2010/abstracts.html#4_1_1 (visited on 09/02/2017).

[25] Sparsh Mittal. “A Survey of Recent Prefetching Techniques for Processor Caches”. In: *ACM Computing Surveys* 49.2 (Aug. 2016), 35:1–35:35. ISSN: 0360-0300. DOI: 10.1145/2907071.

[26] Todd Mowry and Anoop Gupta. “Tolerating Latency Through Software-controlled Prefetching in Shared-memory Multiprocessors”. In: *Journal of Parallel and Distributed Computing* 12.2 (1991), pp. 87–106. DOI: 10.1016/0743-7315(91)90014-Z.

[27] PostgreSQL Global Development Group. *PostgreSQL*. Version 10 Beta 1. 2017. URL: https://www.postgresql.org (visited on 08/23/2017).

[28] Alex Ramirez et al. “Code Layout Optimizations for Transaction Processing Workloads”. In: *SIGARCH Comput. Archit. News* 29.2 (May 2001), pp. 155–164. ISSN: 0163-5964. DOI: 10.1145/384285.379260.

[29] Salvatore Sanfilippo. *Redis*. Version 3.2.9. 2017. URL: https://redis.io (visited on 08/23/2017).

[30] K. Shyamala et al. “Instruction Prefetching Using Basicblock Prediction”. In: *Proceedings of the 2008 International Conference on Electronic Design (ICED)*. IEEE. 2008, pp. 1–4. DOI: 10.1109/ICED.2008.4786750.

[31] Alan J. Smith. “Cache Memories”. In: *ACM Computing Surveys (CSUR)* 14.3 (Sept. 1982), pp. 473–530. ISSN: 0360-0300. DOI: 10.1145/356887.356892.

[32] Alan J. Smith. “Sequential Program Prefetching in Memory Hierarchies”. In: *Computer* 11.12 (Dec. 1978), pp. 7–21. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218016.

[33] Lawrence Spracklen, Yuan Chou and Santosh G. Abraham. “Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications”. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2005, pp. 225–236. DOI: 10.1109/HPCA.2005.13.

[34] Laszlo Szekeres et al. “Sok: Eternal War in Memory”. In: *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[35] Yuan Taur. “CMOS Design Near the Limit of Scaling”. In: *IBM Journal of Research and Development* 46.2.3 (2002), pp. 213–222. DOI: 10.1147/rd.462.0213.

[36] The SystemTap Contributors. *SystemTap*. Version 3.2/0.166. 2017. URL: https://sourceware.org/systemtap (visited on 07/01/2017).

[37] The SystemTap Contributors et al. *SystemTap Language Reference*. Oct. 9, 2015. URL: https://sourceware.org/systemtap/langref (visited on 08/22/2017).

[38] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. 1995. URL: http://refspecs.linuxbase.org (visited on 09/02/2017).

[39] Transaction Processing Performance Council. *TPC Benchmark C*. Version 5.11. 2010. URL: http://www.tpc.org/tpcc (visited on 09/02/2017).

[40] Arjan van de Ven. *New Security Enhancements in Red Hat Enterprise Linux*. Whitepaper. Version 3, update 3. Red Hat, Inc., 2004. URL: https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf (visited on 08/23/2017).

[41] Vish Viswanathan. *Disclosure of H/W prefetcher control on some Intel processors*. Intel Developer Zone. Sept. 24, 2014. URL: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors (visited on 09/03/2017).

[42] Jingren Zhou and Kenneth A. Ross. “Buffering Database Operations for Enhanced Instruction Cache Performance”. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. ACM. 2004, pp. 191–202. DOI: 10.1145/1007568.1007592.