Alleria: An Advanced Memory Access Analysis Framework

ABSTRACT

Dynamic program analysis and simulation tools are used extensively by computer architecture and compiler optimization researchers to improve existing techniques or develop new ones. We propose the Alleria framework to make it easier for researchers to collect interesting information about one or more target applications. The user can reduce the overhead of profiling by specifying exactly those pieces of code that are of interest in a window configuration. This novel type of configuration has been designed to reduce the run-time profiling analysis overhead. It enables the user to specify an event on which Alleria will begin the profiling session, and during the session, it will only instrument those instructions and record information about their execution as specified in the configuration. The user can profile multiple processes at the same time. When the profiling session terminates, a collection of profiles are generated containing accurate and detailed execution traces and other information that can be used to perform analysis or simulation, possibly on a different system. We propose the notion of adaptive profiling, develop and implement a heuristic-based adaptive profiling mechanism, and discuss its potential. We show how Alleria works and how to build tools based on the Alleria framework. In addition, we discuss the performance of Alleria and compare it with similar works. We show that Alleria can be used to profile billions of instructions with a reasonable running time. We also show that the hard disk drive bandwidth constitutes the bottleneck in Alleria's performance. Finally, a use case study of Alleria is presented.

1. INTRODUCTION

The memory hierarchy of a computer system plays a crucial factor in determining its overall performance. The frequency at which the DRAM memory operates is typically much lower than the frequency at which the processor operates and the memory access latency is substantially larger than a processor clock cycle. This leads computer architecture engineers to introduce one or more levels of memory modules between the processor and main memory to avoid the access latency of DRAM. Modern memory hierarchies are becoming increasingly complex and they evolve to perform better with respect to common access patterns that occur in production software applications.

This means that a particular memory hierarchy is usually optimized for specific access patterns. If a running application accesses memory according to these patterns, it will perform significantly better than if it did not. Therefore, the application itself has to be implemented in a way that is friendly to the memory architecture it is running on to benefit from it. In addition, a compiler has to optimize that application to achieve the same goal. The two most common access patterns are temporal locality of reference, in which the same location is accessed multiple times in a small duration of time, and spatial locality of reference, in which nearby locations are accessed in a small duration of time.

The currently dominant main memory technology is Dynamic Random Access Memory (DRAM). This technology suffers from a critical technical issue — density. It is becoming increasingly difficult to fit more DRAM bits in the same area, especially without increasing production cost and/or energy consumption (which in turn causes more heat dissipation) [1, 2]. Persistent memory technologies have been used in the past few years either as a replacement for DRAM (in embedded systems) or together with DRAM [3, 4, 5, 6, 7]. However, due to their special physical characteristics, persistent memory technologies have spawned another class of memory access patterns, namely the pattern of values that are being read and written. That is, the performance and energy consumption of a computer system that includes either only persistent memory or both persistent memory and DRAM modules as main memory not only depend on the addresses of the memory locations that are being accessed but also on the values that are being read or written [8, 9]. Knowing the memory addresses and values that are being read or written is extremely important for other purposes including profile-guided optimization [10, 11] and memory error detection [12]. We discuss some memory error detection tools in Section 7.

By understanding the memory access behavior of production applications, computer architecture researchers can design memory architectures that are tuned to enhance the performance of these applications, compiler optimization researchers can propose more sophisticated compiler optimizations and evaluate their gains and software developers can discover which pieces of code do not exhibit friendly access patterns and can also detect and diagnose errors related to memory access and management. Persistent memory technologies makes this task more difficult.

1.1 Alleria

The Alleria framework has been designed to make it easier for researchers and developers understand the memory

access behavior of multithreaded and/or multi-process production applications to achieve these goals. It implements numerous novel features to enhance the profiling experience and significantly improve the accuracy of the generated profiles. The main component of the Alleria framework is a profiler based on the Pin dynamic binary instrumentation framework [13, 14]. From now on, the term Alleria refers to the Alleria profiler unless otherwise mentioned.

Alleria is purely a profiler and does not perform any simulation or access pattern analysis. The reason for this is that Alleria has been designed to enable trace-driven simulation and analysis rather than execution-driven analysis. Commonly used simulators such as Sniper [15] can be used to perform complex memory access analysis on-the-fly as the target application is running. However, since the analysis has to be performed while the application is running, it can be very slow, especially when analyzing large production applications. Also they may include many features that are not useful for memory access analysis in particular and so they reduce performance unnecessarily. On the other hand, Alleria examines all executing instructions and comprehensively records all memory accesses. The result is a profile containing all the information needed to perform analysis or simulation offline using tools such as Ramulator [16]. This approach has significant advantages in addition to lower overhead [17]. First, since Alleria writes all collected data to profiles on non-volatile memory (such as HDD), it does not have any of the problems associated with shadow memory [18, 19]. In fact, as we will discuss later, Alleria (and Pin) typically allocates a small amount of memory that is configurable. Most tools that use shadow memory can be built as offline memory error detection tools based on Alleria profiles. Second, the simulation or analysis can be performed on a system on which the target application cannot run. Third, the overhead of profiling is incurred only once. Simulation can be then performed easily many times with different computer system configurations as input. However, this introduces the challenge of maintaining massive profiles.

Most existing profilers are designed for specific tasks which can be carried out gradually as the target application is being profiled. When the application terminates, the profiler produces relevant statistics. Since the output of such profilers is of statistical nature, its size is not an issue. On the other hand, non-statistical profilers have a problem regarding how to manage the output profile because it can be very large in size (simulators also have this problem). An example of such profilers is Adept [20], which is briefly discussed in Section 7. In this case, the data emitted to a profile at runtime has to be as compact as possible to reduce the disk I/O overhead and to make it easier to share the generated profiles and consume them by other tools. Naturally, these profiles must have a binary format. Alleria can produce binary profiles that have a novel file format specifically designed for compactness, extensibility and ease of consumption by offline analysis tools. However, producing highly compact profiles is not an objective of this work. We discuss this issue in more detail in Section 8.

There are many existing tools that are used to analyze the memory access behavior of applications. In addition to the simulators mentioned above, there are specialized functional cache simulators such as CMP$im [21] and Cachegrind [22]. Most of these tools currently use virtual addresses rather than physical addresses to perform the simulation and analysis. This makes them not suitable for analyzing production applications or more than one program together. Few use physical addresses but support only fixed address translation mechanisms [23]. Using only virtual addresses has the following shortcomings:

- The majority of caches in real systems are not virtual. That is, they are either physically indexed and physically tagged or virtually indexed and physically tagged. They are rarely fully virtual. So any analysis based on virtual addresses does not actually capture what happens in reality.
- In a single process, more than one virtual page can be mapped to the same physical page.
- It is impossible to simulate or analyze more than one process based on virtual addresses. Different processes have different virtual address spaces. Some of the virtual pages might be mapped to the same physical page (consider sharing libraries or memory-mapping interprocess communication). Of course, other virtual pages might be mapped to different physical pages.
- The physical address of a page may change when the OS swaps the page out and in while the virtual address remains the same. This has implications on the cache that cannot be captured using only virtual addresses.
- Even for a single process with a non-changing memory map, virtual memory can have an unanticipated impact on the energy consumption and timing when using cache simulators depending on the cache organization being simulated [24, 25].
- Studies on side-channel attacks on caches require at least physical addresses or both virtual and physical addresses [26, 27].

Alleria has the ability to record the physical addresses that correspond to the virtual addresses of the accessed memory locations at the time they were accessed with very high accuracy. This enables researchers to understand the memory access behavior of multiple applications running concurrently or applications that consist of multiple processes. While this task might be easy on Linux (most of the tools mentioned above run only on Linux), getting physical addresses on Windows is more tricky.

Another novel feature of Alleria is the ability to periodically record the state of the virtual address space of the target process. This state includes which regions are currently allocated, what they are being used for, which function allocated them, in which thread and when. We believe that this can be very useful when analyzing memory access behavior. Consider the following examples:

- Memory regions that are used for different purposes can and have different memory access behaviors. Therefore, one can focus on specific memory regions and research on how to reduce the time or energy required to access them [28].

- Windows allocates memory at 64 KB granularity. Linux allocates memory at 4 KB granularity. That is, allocating an amount of memory that is not multiple of the allocation granularity causes a phenomenon known as internal virtual address space fragmentation. The application may quickly run out of memory. One has to take snapshots of the virtual address space at run-time to reveal such issues.

VMMap [29] and RAMMap [30] can take a periodic snapshot of the virtual address space of a selected process and the physical address space of the system, respectively. Alleria does the same except that it augments these snapshots with information generated by instrumentation.

Alleria has the ability to record many different pieces of information about the executing program including dynamic instructions, virtual addresses, physical addresses, accessed values, thread scheduling information, and function names. Enabling a large number of these features for all instructions can substantially increase the profiling overhead. On large applications, when enabling all features, the performance overhead of Alleria can be thousands of times larger than the native performance, rendering it inconvenient. This problem is not specific to Alleria, but common between most instrumentation-based profilers. This overhead can be reduced very specifying precisely the regions of code that are of interest. Existing profiling tools and simulators are very limited in their support for specifying regions of interests (ROIs).

Many existing tools offer numerous switches that can be used to turn on or off specific features of the tool during the whole period of running the target application. However, they offer very little control over when to profile and what to profile. A software developer, for example, is typically only interested in understanding the memory access patterns of the code that he or she has written excluding the code that is part of the operating system or third-party libraries. A researcher might be interested in only the behavior of the application during its actual processing excluding startup and termination. To solve this problem, we introduce another novel feature of Alleria called Window Configuration.

Window configuration enables the user to specify windows in time during which Alleria will profile the application. A window opens when a specified event is triggered during the execution of the application and closes when some other event is triggered at some time later. As a simple example, one can tell Alleria to start profiling when a function called *Foo* is called for the first time and stop profiling when *Foo* returns. When there is no open window, the overhead of Alleria is negligible. The user can also specify what to profile during that window. For example, the user can specify to profile only the *Foo* function and not any function that it calls.

So far we have only talked about the Alleria profiler. The following list summarizes the goals of the Alleria framework.

- Portability: The Alleria framework should be designed so that it can be easily ported to different operating systems. We have implemented Alleria fully on Windows and partially on Linux.
- Efficiency: Alleria tries to do minimum amount of work just enough to record the requested information. Application threads should run as fast as possible.
- Extensibility: The Alleria profile binary format is extensible. That is, it allows other tools to insert more information in the profile easily without corrupting the profile. An analysis tool can only consume the parts of the profile that it has been designed to consume and ignore everything else.

The Alleria profiler focuses on instructions that access memory. It records not only the addresses of the locations being accessed, but also the values being read or written. To achieve this, it has to have an intimate knowledge of every instruction that accesses memory. The values are required for a variety of tasks such as cache compression and page de-duplication. At the time of writing this paper, Alleria supports the full x86 and x64 instruction set up to and including the Intel Broadwell microarchitecture.

Alleria is suitable for the following purposes:

- Understanding the memory consumption of applications.
- Building trace-driven cache and memory simulators that are fast and accurate and suitable for architectural security studies. More generally, almost any architectural micro-simulator can be built [31].
- Detecting and investigating memory errors such as memory corruption.
- Understanding cache and memory access patterns [32, 33, 34, 35, 36, 11, 10] and measuring reuse distances [37]. A trace-driven tool can analyze Alleria profiles and guide an optimizing compiler to perform profile-guided optimizations.
- Analyzing memory access patterns per memory region [38].
- Discovering and analyzing security vulnerabilities.
- Building cycle-accurate architectural trace-driven simulators. Even though Alleria does not capture values of registers at every instruction, those values are not generally necessary for architectural simulation as long as the output of the application being simulated is not needed (which is typically the case).

One particular example of a tool that can be much easier built on top of Alleria is LDoctor [11]. LDoctor is a sophisticated tool that profiles the behavior of loops to detect a variety of performance issues such as redundant computations. Then these issues can be fixed either manually at the source code level or through profile-guided compiler optimizations. The tool works by instrumenting only specific methods. Within the selected methods, only those instructions that read from a particular memory region, namely the heap, and that exist in a loop are profiled. In addition, LDoctor requires the values being read and not the addresses of the locations being read from. Alleria makes it considerably

easier to develop tools with such requirements. Another example is the REDSPY value locality profiler [10] (especially profiling spatial value locality).

The contributions of this work are the following:

- We develop a fine-grained profiling framework with emphasis on memory accesses.
- We propose an adaptive mechanism to efficiently use multiple threads to collect and record information about multithreaded applications.
- We argue that physical addresses are important and discuss how to capture them.
- An XML-based configuration that enables the user to control the profiler at different levels (process, thread, library, function, and instruction).
- A technique that enables the user to select processes to be profiled from a process tree.
- An evaluation of the profiling throughput of Alleria.

We discuss the design and implementation of the Alleria profiler in Section 2. Adaptive profiling is discussed in Section 3. We explain our validation methodology for Alleria in Section 4. Performance evaluation is discussed in Section5. An simple use case of Alleria is demonstrated in Section 6. Then, in Section 7, we discuss similar tools and related work. In Section 8, we provide additional discussion on the design of Alleria. Finally, we conclude and discuss future work in Section 9.

2. DESIGN AND IMPLEMENTATION

Now we present an overview of the architecture of the Alleria profiler and provide a description of how it works. Figure 1 shows the main components of Alleria. Since Alleria is a pintool, it is built as a shared library. When the Pin launcher is run, it first creates another process in a suspended state with the target executable as the main executable for that process. Then it injects the Pin VM shared library and the Alleria shared library, which in turn initializes itself and tells Pin VM to start running the application.

For every shared library (also known as an image) that is loaded in the process, Pin notifies Alleria of the library. Alleria internally maintains a list of all loaded libraries and all functions defined in these libraries (if debug information are available). This information is used during instrumentation to selectively profile functions and is also used when emitting the resulting traces.

The Live Lands Walker (LLW) is responsible for recording the state of the whole user-mode virtual address space of the process. This state includes the base addresses and sizes of the allocated regions and what they are being used for (stack, heap, executables, memory-mapped files, and raw). However, it does not say anything about who is using these regions. This is extremely important because the application is not the only program that is allocating memory, Pin and Alleria also allocate many memory regions. For example, Pin allocates memory to maintain shadow stacks and the code cache. Alleria requires memory mainly to store partial traces temporarily. One should be able to filter these memory regions and restrict any later simulation or analysis to only application memory regions.

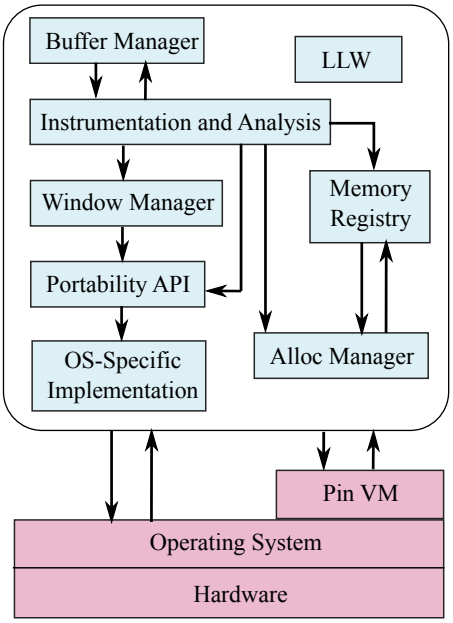

Figure 1: Alleria profiler architecture. Arrows indicate dependence where the component at the source of an arrow depends on the component at the destination of the arrow. Some components and dependencies have been omitted for simplicity.

To solve this problem, we have implemented an allocation manager. All memory allocated by Alleria is requested from the allocation manager, which in turn allocates memory from the heap managed by the statically-linked C runtime, records that such an allocation has happened and returns a pointer to the allocated block. The memory registry keeps track of all allocations that are issued from Alleria and the application and some of the allocations from Pin VM. Allocations from the application (the main executable and any libraries it uses including operating system libraries) are intercepted by Alleria to record them in the memory registry. Pin VM only reports allocations related to the code cache. Therefore, while the LLW can determine all allocated memory regions, it will not be able to know who is using some of them. Practically, it can know the users of more than 90% of all memory regions. The rest might be either used by the operating system or the Pin VM. Those regions with unknown users are definitely not used by the app and therefore this does not significantly affects the usefulness of this information.

The LLW is completely OS-dependent and so it does not depend on the Portability API. We have implemented an LLW for Windows and another for Linux. Similarly, virtual to physical (V2P) address translation is completely OS-dependent and it does not use the Portability API. However, this particular feature requires that the process has been granted Administrator (root on Linux) privileges. Therefore, if the user enabled V2P translation, the process has to run with Administrator privileges. Otherwise, Alleria emits an

error and terminates the process. If the user disabled V2P translation, then neither Alleria nor Pin VM require administrator privileges and therefore the process can be created without them unless required by the application itself.

2.1 Multi-Threaded Profiling

In large applications, during a profiling session, billions or trillions of instructions might get executed. If we only have one thread-safe file stream to which textual traces are being written [39], then writing to this stream is going to be the major bottleneck in the profiler. Even if that file stream uses a large internal buffer to be used before writing to the disk drive, analysis routines would be too large and complex to be optimized by Pin VM's JIT compiler. Alleria employs a number of optimizations that significantly reduce the overhead of profiling and shifts the bottleneck to the persistent memory hardware.

Alleria uses an important feature of Pin that enables it to create processing threads. These threads are completely managed by Alleria and are not subject to any instrumentation as opposed to app threads. Alleria accepts a command-line switch that specifies the number of processing threads to create. Typically, this number is proportional to the number of compute-bound app threads. The idea of using processing threads to improve profiling performance has been used in previous works [40, 41].

When Pin VM initializes Alleria, it creates that many processing threads. Each of these threads has its own file stream to which it writes traces. Since this file stream is used by only one thread, it does not have to be protected by a lock and it can operate at full speed. In addition to the processing threads, Alleria creates two special-purpose processing threads. One of them is used to notify the window configuration manager (discussed later) of the elapsed time since the first instruction of the application was executed. The other thread is the LLW thread. The rest of the threads that are running in the process are all app threads and zero or more Pin VM threads. This is shown in Figure 2.

Alleria instruments application code at the trace level. A trace of instructions is defined by Pin as a straight-line sequence of instructions that has exactly one entry point and exactly one unconditional exit point. It might include zero or more conditional control transfer instructions. Overall, the total number of instructions in a trace is limited and typically does not exceed few dozens of instructions.

If a profiling window is open, Alleria examines every instruction in the trace and if it matches the window's profile specification, the necessary analysis routines are inserted before and/or after the instruction. For example, if this instruction transfers a 64-bit value of a register to a memory location, Alleria inserts an analysis routine before this instruction to record the memory access address and size and an analysis routine after the instruction to record the value that was written. Other data is being recorded too. Analysis routines might also be installed before and after a function gets called. Note that functions may exit abruptly either because of processor generated exception or a system call that did not return to the function. We consider these cases and make sure that the profile being recorded does not get corrupted when they occur and that the execution of the app continues

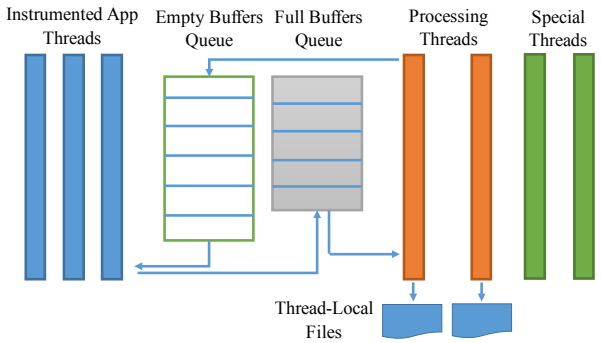

Figure 2: The threads that are running during profiling and the state they are maintaining. The three special threads includes a Pin VM thread, the timer thread and the LLW thread. Note that the number of app threads can be one or larger and the number of processing threads can be zero or larger.

as if it was not being profiled. This is performed by tracking such control flows. Analysis routines can be much more complex than what is described here depending on the instruction being analyzed (such as the vector scatter instruction set).

All of this data is stored temporarily in a fixed-size thread-local buffer. By thread-local here we mean local to an app thread. Just like with file streams, since these buffers are used by one thread, there is no need for a lock and accesses to them are executed at full speed. In addition, there would be no need to associate a thread an identifier with each record in the buffer. Pin offers a fast buffering API [42], but we decided not to use it so that we can use the same buffer to store multiple different types of objects. This allows us to fully utilize the buffers without wasting space.

Eventually, the buffer an app thread is using will get full. This buffer is moved to a global queue of full buffers. Then the app thread checks a global queue of free buffers to see whether a free buffer is available. If that was the case, it acquires the free buffer and continues running. Otherwise, it waits until a free buffer is available. There is a special case, however. If there are no processing threads running at the time the current buffer becomes full, the app thread itself processes the buffer by writing it to a global file stream (not shown in 2). Otherwise, there exists at least one functional processing thread and so the app thread simply waits for a free buffer to become available.

We mentioned earlier that Alleria instruments application code at the trace level. This has the advantage of lower instrumentation overhead. However, it has also a disadvantage. Since not all instructions in a trace might be executed, slots in the buffer might remain unused and have to be recognized and skipped while dumping the buffer to the file. We will evaluate this internal wastage in Section 5.

Determining the ideal number of processing threads, the ideal number of buffers per app thread and the ideal size of a buffer is not trivial. We explore how these parameters affect the running time of app threads and the efficiency of processing threads in Section 5.

2.2 Live Lands Walker (LLW)

One of the special threads shown in Figure 2 is the LLW. The LLW can be disabled using a command-line switch. Alleria creates the thread at initialization time. The thread executes a loop that iterates through the whole address space, recording all free and allocated regions and their attributes (size and protection). Then it goes to sleep for a configurable amount of time. This is the general way that both Windows and Linux offer to capture the address space and it suffers from two potential issues. First, on Windows, without suspending all the other threads, the state of the address space may change while the LLW is walking it, resulting in a snapshot that is not perfectly consistent. This is mostly not an issue because it is difficult to define what it means for a snapshot to be consistent for a multi-threaded application. Even for a single-threaded application, such accuracy is generally not required as long as the walker does not take a substantial amount of time. The performance of the walker depends on the sizes of the free regions since these regions are empty and need not be walked in contrast to allocated regions. The larger the free regions, the quicker the walker finishes. The second potential issue is the frequency of walking the address space. A frequency of one second or more is usually sufficient, although that is configurable.

2.3 Capturing Physical Addresses

On Windows, the only way to obtain the physical address of a given virtual address starting with Windows Vista and Windows Server 2008 is to be in kernel-mode [1]. Therefore, Alleria uses a kernel-mode software driver that simply uses a single kernel-mode API to perform this task. The driver takes an array of page-aligned virtual addresses as input, calls that API to get the physical address and returns an array of the corresponding physical addresses. On Linux, this can be done without resorting to kernel-mode APIs by using the pagemap file of the proc pseudo-filesystem.

However, this process is a little fragile on both Windows and Linux because the OS may choose to swap out a page before we get its physical address. This causes the profile to fail converting some virtual addresses. For missing physical addresses, the last captured physical addresses can be used or the virtual addresses can be omitted from analysis. The OS might also swap out a page and swap it back in to a different physical address causing the profiler to capture a different physical address. Fortunately, these cases do not happen frequently. The OS will only swap out a page when it is under severe memory pressure. If that happened, Alleria will not be able to distinguish between the different physical pages that were mapped to the same virtual page. Therefore, correct usage of the physical addresses by a tool based on Alleria requires knowledge of the size of the working set of the application.

2.4 Capturing Thread Schedules

Knowing which core a thread was running on can be useful for simulation and performance analysis. A schedule of threads has a substantial impact on the effectiveness of the cache hierarchy and simultaneous multithreading (SMT).

[1]There is an undocumented user-mode API that can be used to take a snapshot of the whole memory map of a given process but that would be inefficient for mapping specific virtual addresses.

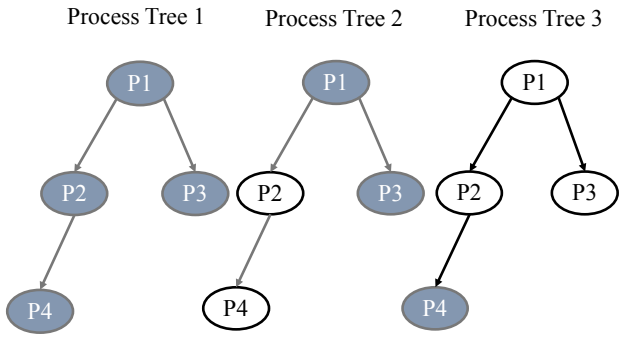

Figure 3: Example process trees. Ellipses with gray background color represent processes that the user wants to profile. Ellipses with white background color represent processes that the user does not want to profile.

Linux and Windows employ sophisticated preemptive thread scheduling algorithms. It is generally difficult to accurately predict beforehand on which core a thread will run. In addition, these operating systems do not offer synchronous or asynchronous notifications when a thread is scheduled to run on a particular core. For these reasons, it is impossible for a profiler to capture precisely and fully the thread schedule of a process.

However, when a thread is running on a particular core, it is likely that it will be scheduled to run on the same core in the next quantum. Moreover, Linux offers APIs that enable us to capture the current thread assignment for all threads, while Windows offers APIs that can be used to determine on which core the current thread is or was running on. Alleria uses these APIs to capture thread scheduling information. This is done at the Pin trace granularity. Alleria instruments the beginning of each trace in each app thread to obtain the identifier of the current core. If a thread changed the core it is running on multiple times during the execution of the same trace, only the first and last cores will be captured by Alleria. This situation is unlikely to occur because a trace is typically only a few dozen static instructions. A trace may only contain a simple loop (does not contain function calls or system calls) such that it is likely that the thread on which it is executing will be scheduled on the same core during the execution of that loop.

2.5 Profiling Multiple Processes

Windows does not allow loading an executable binary dynamically in a process that is already running (it can be loaded as a data file, though). On Linux, this is possible as long as the executable was compiled with -fpie and -rdynamic compiler switches (which produces a binary that is relocatable). Therefore, if an application wants to access a feature offered by another application, it may need to create another process.

It is not uncommon for large applications to create other processes to request a service from an executable. Probably the most familiar example is shell programs (bash on Linux and cmd on Windows). Another example is when a

web browser creates a process in which a plug-in will play a requested video.

To our knowledge, the Alleria profiler is the first to offer a fine-grained multi-process configuration. We will discuss the configuration part in the following subsection. In this subsection, we will discuss the design and implementation.

To enable the user to have control over which processes to be profiled, we need to solve two problems. First, every process needs to have an identifier that the user can use in the configuration file to determine which process to profile (or not to profile). Operating systems identify processes using a value commonly called pid. However, since this value is assigned to a process at the time it is created, it is not possible for the user to know it beforehand and therefore use it in the configuration file. Using the file path of the executable leads to an ambiguity in case there are multiple instances of the same program. So we need another identification system. Second, we need a mechanism to correctly assign identifiers to processes without duplications.

Let P1 be the process that is the first one to be injected with Alleria (either by attaching Pin to it or by having Pin to create the process). We call this process the genesis process. We also refer to any processes that P1 creates or forks as the child processes. A child process might in turn create other processes. Even in this case, that child process will not be considered as a genesis process. We refer to the whole hierarchy of created processes as the process tree. Alleria supports profiling more than one process tree (that is, unrelated processes) concurrently.

To solve the first problem, we exploit the order in which processes are being created. Practically, an application creates a process either in a deterministic way or in response to some user input. Either way, the user can easily determine the order in which processes are created by running the application once without profiling. We use this order to identify processes to be profiled. The order at which processes terminate (whether they crashed or not) does not matter.

Figure 3 shows three examples of process trees. Consider the tree on the left of the figure. The genesis process is P1 and always has an identifier of 1. When P1 is about to create another process, it increments a counter that is global with respect to the process tree. In this example, the result is 2 and this value is considered to be the identifier for P2. Now suppose that P1 creates another process, namely P3. The counter is incremented and P3 will have an identifier of 3. Similarly for P4, its identifier will be 4. In this example, we are assuming that the user has specified in the configuration file to profile any process.

The process tree in the middle of the figure shows an example where the user has specified to not profile P2. This means that Alleria will not be injected in P2. Consequently, Alleria will not be aware of any processes created by P2 and the whole subtree rooted at P2 will not be profiled. However, P2 will still have an identifier, namely 2. So the identifier of P3 is 3 and any child processes of P2 have no identifiers and do not have any effect on the process tree counter.

The last process tree in the figure shows a slightly more complex case in which the user is only interested in P4 and does not want to profile any of the other processes including the genesis process. We show how to easily achieve this in the next subsection.

Now we discuss the second problem, namely assigning identifiers to processes. This is accomplished by using a shared memory region that stores a data structure called the parent process number (PPN) database. This memory region is not only shared by a process tree, but by all processes that are currently being profiled by Alleria on the same operating system. All process tree counters are stored in this region and updated in a thread-safe fashion using an inter-process mutually exclusive lock. The memory region is created by the first process that is profiled by Alleria and destroyed by the last profiled process, irrespective of the process tree to which they belong. The implementation makes heavy use of the Portability API from Figure 1.

2.6 Window Configuration Overview

If we run a typical large application interactively for few minutes, the total number of user-mode instructions that would have been executed would probably be in tens or hundreds of billions. Profiling all of these instructions with all features of Alleria turned on is going to take a significant amount time (hours or days). We will discuss the overhead of Alleria in Section 5. However, in this section, we will present a novel type of configuration that can help reduce this overhead, sometimes, considerably.

Window configuration is an XML configuration designed specifically for profilers. It is named so because it enables the user to define windows of profiling in time. A window is a configuration element that consists of three elements:

- Zero or one On event element.
- Zero or one Till event element.
- Exactly one Profile element.

We will explain these elements with an example. Listing 1 shows a simple example of a window configuration file that defines one window with an On event, a Till event and the Profile element.

The On event specifies to the profiler when to start profiling. This can be based on functions calls or returns, libraries loads or unloads, instruction counts and other events. It is also possible to specify a combination of events. The On event specified in this example simply states that the window opens when the function whose compiler-mangled name is main and defined in the process's main executable is first called in any thread. This implies that the profiler will start profiling from the first instruction of that function the first time it is called.

The Till event specifies to the profiler when to stop profiling. This can be based on the same events as the On event. In this example, the Till event states that the window closes when the function whose compiler-mangled name is main and defined in the process's main executable returns the first time in any thread. This implies that the profiler will stop profiling just after the last instruction of that function completes its execution.

The On event and the Till event do not fully describe to the profiler what to do. They just describe a window in time in which the profiler should be active. However, they do not say anything about what to profile about the instructions that

will be executed during that window. That is the purpose of the Profile section.

The Profile section has a detailed hierarchy that reflects the multi-process profiling environment. A Profile tag contains one or more Process tags. A Process tag provides profiling configuration regarding that process only.

```
<AlleriaConfig version="1.0">
    <Window>
        <On>
            <Call name="main" />
        </On>
        <Till>
            <Return name="main" />
        </Till>
        <Profile>
            <Process>
                <Thread>
                    <Image main="true">
                        <Function>
                            <Instruction
                                cat="anyMem"
                                routineId="
                                    true"
                                timeStamp="
                                    true"
                                accessValue=
                                    "true"/>
                        </Function>
                    </Image>
                </Thread>
            </Process>
        </Profile>
    </Window>
</AlleriaConfig>
```

Listing 1: A window configuration file sample.

In the example shown in Listing 1, the Process tag has no identifier and therefore it represents all processes in the process tree. A Process tag contains zero or more Thread tags and a Thread tag contains one or more Image tags. We will explain why a Process tag may contain zero Thread tags shortly.

In the example, the Process tag contains one generic Thread tag. The Thread tag contains one specialized Image tag. It is specialized because it specify a particular image — the main executable in this example. In the Image tag, there can be one or more Function tags, each can contain one or more Instruction tags. In this example, the Function tag specifies a generic function and the Instruction tag specifies any instruction that accesses memory. The Instruction tag also specifies the information to record about each instruction.

Now we can summarize what the example configuration file tells Alleria to do. It says to start profiling when the main function gets called and to stop profiling when it returns. While code is being executed from the main executable in any process in the process tree and in any thread, record every instruction that accesses memory.

The reason why a Process tag may contain zero Thread tags is to support some configurations such as the one described by the rightmost process tree in Figure 3. In that process tree, the user is only interested in profiling P4. However, for Alleria to be aware of P4, it has to be injected in P1's child processes. At the same time, P1 and its child processes should not be profiled. This is extremely important to reduce the total performance and memory overhead on the system. To achieve this, we can specify two Process tags under the Profile tag. The first is a generic and empty Process tag. It has to be generic to inject Alleria in P1's child processes. It has to be empty to not profile anything. The second Process tag identifies P4 and contains the required sub-tags to profile that process.

2.7 Interceptors

An interceptor is a powerful instrument that intercepts calls and returns to a particular function for the purpose of capturing the arguments and return values. Listing 2 shows how to specify an interceptor for a standard C main function in window configuration. In the Types section, types of arguments and return values can be defined. A standard C main function has two parameters. The first parameter is of type int and the second is of type char*[]. Window configuration has built-in support for a number of primitive types such as integers and floating-point numbers. However, a type that represents an array of strings need to be defined by the user. The type named string represents an array of 1-byte characters with a terminating sentinel of null. An argument of this type points to the element of the array with the smallest address. The type named strings is an array of strings where the number of elements is given by an argument called argc. In the Function, the name of the function main has been specified and an Intercept section is given. The Intercept section specifies the return type, the names, types, and order of the parameters of interest. Not all parameters need to be specified. Only those of interest can be specified. The user can specify for each parameter whether it acquires its value as input, output, or both. Note how these parameters match those of the C main function.

```
<AlleriaConfig version="1.0">
  <Types>
    <Pointer name="string" elementType="
        ascii" direction="inc" sentinel="0"
        />
    <Pointer name="strings" elementType="
        string" direction="inc"
        countVariable="argc" />
  </Types>
  <Window>
    <Profile>
      <Process>
        <Thread>
          <Image>
            <Function name="main">
                <Intercept retType="s4">
                  <Parameter name="argc"
                      index="0" type="s4" in
                      ="true"/>
                  <Parameter name="argv"
                      index="1" type="
                      strings" in="true"/>
            </Function>
          </Image>
        </Thread>
      </Process>
    </Profile>
  </Window>
</AlleriaConfig>
```

Listing 2: A window configuration specifying an interceptor.

When Alleria parses the configuration file, it intercepts all calls and returns to the main function and captures the values of the parameters and return values. Even though the argv parameter is a pointer, it will be captured together with whatever it is pointing to.

Interceptors are particularly useful for profiling pure functions, which are common in functional programming. A pure function is a function that does not read from any non-local locations other than the arguments passed to it and does not write to any non-local locations other than the return value location. Interceptors would record only the arguments and return values function calls without profiling the instructions of the function itself. This allows us to let the function get executed at native speed without instrumentation while at the same time, being able to regenerate the memory trace of the function by executing it again. The instructions that constitute the function body are recorded as well. This makes the function effectively a small program by itself that can be profiled or analyzed independently.

The interception engine of Alleria manages instrumentation at the function granularity and the buffers used to hold the captured arguments and return values. When an app being profiled terminates, Alleria dumps the contents of the buffers to a profile called the interceptors profile that can be consumed by trace-driven tools. Even though only the arguments, return values, and the function body have been recorded, they are sufficient to correctly execute the function separately because the function is pure. Interceptors can improve the performance of Alleria and reduces the size of generated profiles without compromising the completeness of the profiles.

Consider a function that adds up all the elements of a given array and returns the sum without modifying the array. In this example, using interceptors can substantially improve performance by only capturing the input vector and avoiding instrumentation of the function. However, the size of the recorded trace would be approximately the same since each element is accessed only once. Consider another function that multiplies each element of a given array by a randomly generated number and stores the result in the same array. In this case, the array parameter can be specified in the configuration file as input and output so that it will be captured at the entrance and return of the function. In this example, similar to the previous one, performance can be improved but the size of the generated trace would be the same. Finally, consider a function that multiplies two two-dimensional matrices and stores the result in a third matrix. All the three matrices are parameters of the function. The first two parameters are input and the third is output. Since each element needs to be accessed many times to multiply the matrices, using interceptors can not only improve performance but also substantially reduce the resulting profile size.

A trace-driven tool can use the interceptors profiles in one of the following ways:

- Only use the captured arguments and return values. For example, if the purpose of the tool is to test the function. The captured return value can be compared with the expected return value.
- Use the captured arguments and return values and perform static analysis on the binary code of the function.
- For pure functions, given only the captured arguments, the full memory access trace of the function can generated.

In both the second and third usage scenarios, the tool has be familiar with the same ISA as the profiled program. If the tool is not running on a platform with the same architecture, it will have to emulate the function.

2.8 Binary Profiles and The AbpHelp library

Dumping all recorded information about the executing application in a textual format enables the user to quickly inspect the execution trace without using any additional tools. However, the problem with this is that emitting textual information is extremely expensive in terms of disk I/O activity and the size of the generated profile. For example, the size of a 64-bit address in binary is no more than 8 bytes. On the other hand, it might require up to 16 bytes to represent it in ASCII characters. To this end, we designed binary profiles that are typically 3x smaller than the corresponding textual profiles. In addition, by using the LZMA2 compression algorithm (offline, after generating the profile), a binary profile can be made 8x smaller. This makes it much easier to share using stock portable storage devices and through the Internet. Tools can access the information contained in binary profiles by using a library called AbpHelp. The AbpHealp library has been designed to achieve the following goals:

- Ease of use: The user of the library does not have to know anything about the format of the binary profile or how Alleria or Pin works.
- Portability: The AbpHealp library runs on Windows and Linux and supports both the x86 and x64 architectures.
- Efficiency: The library internally creates data structures that contain potentially frequently accessed information about the profile. So when accessing this information multiple times, no disk I/O will be required.
- Powerful: The library enables the user to open and access many profiles at the same time. This is especially useful when building analysis tools to analyze multiple processes.
- Development flexibility: The library is completely written in C. This makes it lightweight at run-time and consumable from most other programming languages. The AbpHelp library can be used to build powerful analysis tools and simulators that consume binary profiles emitted by Alleria. The library can also be used to develop tools that convert from the Alleria binary profile file format to a format that can be recognized by existing tools.

2.8.1 The HistoBin Analysis Tool

```
#include <iostream>
using namespace std;

#include "AbpApi.h"

UINT64 zeroesRead = 0;
UINT64 zeroesWritten = 0;
UINT64 OnesRead = 0;
UINT64 OnesWritten = 0;

void IncrementStats(PBINARY_PROFILE_SECTION_GENERIC profileEntry);
UINT8 PopCnt(UINT8 byte);

int main(int argc, char *argv[]) {
  if (argc != 2)
    return -1;

  PROFILE_HANDLE profile = AbpOpenProfile(argv[1]);
  if (profile == INVALID_PROFILE_HANDLE)
    return -1;

  BOOL error = FALSE;

  UINT32 secDirSize = AbpGetProfileSectionDirectorySize(profile);
  for (UINT32 i = 0; i < secDirSize; ++i) {
    SECTION_DIRECTORY_ENTRY entry;
    if (!AbpGetProfileSectionDirectoryEntry(profile, i, &entry)) {
      error = TRUE;
      break;
    }

    if (entry.type == BINARY_SECTION_TYPE_PROFILE) {
      BINARY_PROFILE_SECTION_GENERIC profileEntry;
      if (!AbpGetProfileEntryFirst(profile, i, &profileEntry)) {
        error = TRUE;
        break;
      }

      IncrementStats(&profileEntry);

      BINARY_PROFILE_SECTION_GENERIC profileEntryNext;
      while (AbpGetProfileEntryNext(profile, i, &profileEntry, &profileEntryNext)) {
        IncrementStats(&profileEntryNext);
        AbpDestroyProfileEntry(&profileEntry);
        profileEntry = profileEntryNext;
      }
      AbpDestroyProfileEntry(&profileEntry);
    }
  }
  AbpCloseProfile(profile);
  return error ? -1 : 0;
}
void IncrementStats(PBINARY_PROFILE_SECTION_GENERIC profileEntry) {
  if (profileEntry->type == BINARY_PROFILE_SECTION_ENTRY_TYPE_MEMORY_0) {
    PBINARY_PROFILE_SECTION_MEM mem = (PBINARY_PROFILE_SECTION_MEM)profileEntry->entry;
    UINT8 temp = 0;
    if (mem->memopType == MEM_OP_TYPE_LOAD || mem->memopType == MEM_OP_TYPE_STORE) {
      UINT32 index = mem->size;
      while (--index >= 0) {
        temp = PopCnt(mem->info[index]);
        if (mem-> memopType == MEM_OP_TYPE_LOAD) {
          OnesRead += temp;
          zeroesRead += 8 - temp;
        }
        else {
          OnesWritten += temp;
          zeroesWritten += 8 - temp;
        }
      }
    }
  }
}
```

Listing 3: The HistoBin analysis tool written in C++.

We discuss a few APIs from the AbpHealp library and demonstrate a simple example of how to use these APIs to make use of some of the information that Alleria profiles contain. The purpose of the HistoBin tool is to count the number of zeroes and ones that are read or written. The source code of this tool is shown in Listing 3. Note that it contains less than 100 lines of code including error handling, yet it can analyze most x86 instructions. The code can run on practically any operating system and any architecture. This makes it extremely efficient and convenient.

The code consists of three functions: main, IncrementStats and PopCnt. The header file AbpApi.h is the only header file that AbpHelp exports. The main function accepts a path to a binary profile as command-line argument. It calls the AbpOpenProfile function to open the profile and parse its header.

Once a profile is open, a handle to the profile is returned. This handle represents the profile and must be used to access the data it contains. To access memory traces, one has to determine which section in the profile that is of interest. The code shown loops over all sections that contain memory traces and for each of them, it iterates over the recorded memory accesses using the functions AbpGetProfileEntryFirst and AbpGetProfileEntryNext. For each memory access, the value that was read or written is located to count the number of ones and zeroes it contain. One or more global counters are incremented accordingly. Note that the PopCnt function is not shown in the listing and its purpose is to count the number of bits that are set in a given byte.

3. ADAPTIVE PROFILING

Previous works showed that multi-threaded profiling can significantly reduce profiling overhead [40, 41]. However, the problem of how to choose the number of processing threads has not been addressed.

Problem Statement for Reinforcement Learning

There are three parameters that the user has to specify: the number of processing threads, the number of buffers, and the size of a buffer. These parameters can have a significant impact on profiling throughput and so they have to be chosen carefully. However, it is difficult for the user to choose these values without extensive experimentation for every application that is going to be profiled. It is unlikely that users will do that and rather stick with the default values, whatever they are, which may or may not lead to a high profiling throughput. Moreover, the same application may behave differently throughout its execution and the parameters may need to be dynamically adjusted accordingly. A third situation in which adaptive profiling is useful is when profiling multiple processes, each with different behavior. Finally, the same values for these three parameters may result in different performances on different machines and operating systems.

We propose a heuristic-based adaptive profiling mechanism to deal with these issue. To our knowledge, we are the first to do that. Alleria can keep track of certain metrics that are good indicators of performance and then tune these parameters accordingly on-the-fly during profiling. If the behavior of the application changes at some point during execution, the parameters should be tuned accordingly. Our proposed technique, illustrated in 4, uses a number of heuristics to determine how to adapt.

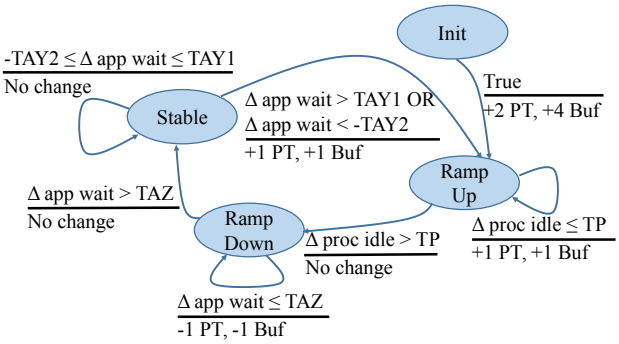

Figure 4: Adaptive profiling finite state machine. The Ramp Up state represents a phase of type X. The Stable state represents a phase of type Y. The Ramp Down state represents a phase of type Z.

The mechanism is described in terms of actions to be performed in response to certain events. We define four types of events. The first event type, initialization, occurs once when Alleria itself initializes, which happens before executing a single instruction of the application. The action to be performed on that event is creating two processing threads and four buffers. The second event type occurs when a new application thread is created, which can happen multiple times during execution. On that event, Alleria creates one buffer.

The third event type is a periodic timer interrupt (e.g., every second) that occurs during a phase of type called X. After initialization, an X phase starts. The total processing thread idle time is a heuristic used to determine when X ends and whether to create a processing thread or not. It is defined as the total amount of time across all processing threads spent waiting for buffers to process. If it increases by at least some specified percentage TP (e.g., 30%) compared to the last occurrence of the event, the phase ends by switching to a phase of type Z. Otherwise, one processing thread and one buffer are created and the phase continues.

The fourth event type is a periodic timer interrupt that occurs during a phase called Y. The total application thread wait time is a heuristic used to determine whether to create more processing threads and buffers or whether to suspend processing threads. It is defined as the total amount of time across all application threads spent waiting for free buffers. If it increases by at least some specified percentage TAY1 or if it decreases by at least some specified percentage TAY2, one processing thread and one buffer are created and an X phase is started during which Alleria tries to adapt to the different behavior of the application. Phase type Y represents a stabilization point — no change in behavior has been identified and therefore there is no need to adapt.

Creating too many processing threads may adversely impact performance. In an X phase, if, after creating a processing thread, performance continues to deteriorate, it means that there are too many of them. Phases of type Z are responsible for reducing the number of active threads. The application thread wait time by itself cannot be used to determine whether to go to an X phase or a Z phase. That

is why a Y phase has to switch first to an X phase to determine what to do to improve performance rather than directly switching to a Z phase. The fluctuation in application thread wait time is bounded by the available hardware resources including computing resources (cores) and I/O bandwidth. If the application being profiled itself is using one or more of these resources intensively, Alleria will be competing with it to use those resources, which will become the bottleneck. When that happens, Alleria will, in a Z phase, suspend some threads and make more resources available for the application until they are no longer needed. In summary, there are two cases in which TAY1 is surpassed: the application's activity increases and therefore more threads need to be created, and the application contends with Alleria on resources and therefore some threads need to be suspended. There is once case in which TAY2 is surpassed: the application's activity decreases (thereby potentially making more resources available of Alleria) and therefore creating more threads could be beneficial.

Note that the size of a buffer is not being tuned, rather it is fixed. Also note that processing threads are never terminated and allocated buffers are never deallocated. If there is more of them than necessary, they will either be idle or suspended. We provide an intuition of why this technique works in practice and evaluate it in Section 5.3 and illustrate it in Section 5.4.

Phases X and Z are designed to adapt to the behavior of the application, while phase Y is designed to settle or trigger that process. The thresholds TP, TAY1, TAY2, and TAZ can be selected empirically so that the phases behave as intended. If the wait time threshold was too small, it can get triggered very frequently potentially resulting in a processing thread bloat or scarcity. If it was too big, there will be little adaptation. Similarly, if the idle time threshold was tool small, the technique will fail to adapt quickly. If it was too big, it may never settle. We found experimentally that suitable values for TP range between 50-100% and suitable values for TAY1, TAY2, and TAZ range between 1-10%.

4. VALIDATION

We validated Alleria using four methods. First, we profiled simple well-understood programs that perform sorting, searching, and matrix multiplication using Alleria with all features turned on. The generated profiles were analyzed using AbpHelp to make sure that they are identical to the actual behavior of the programs and no memory accesses or instruction were missed or corrupted. Second, numerous benchmarks were profiled using Alleria (as discussed in Section 5) and the programs emitted outputs identical to those when running them normally. Third, we wrote small programs that use specific x86 instructions to make sure that Alleria profiles every kind of instructions correctly. Fourth, we were able to successfully profile a number of production applications with graphical user interfaces (GUIs) by providing them with input through the GUI and noting that they behaved identically to their normal functional behavior.

5. EXPERIMENTAL EVALUATION

5.1 Setup

All the experiments were performed on a dedicated Dell Precision Tower 7810 workstation. The specifications of the machine are described in Table 1. We use the single-threaded SPEC CPU2006 suite [43] and the multithreaded GraphBig CPU benchmarks [44]. All the benchmarks were compiled to 64-bit binaries using the Microsoft Visual C++ compiler version 19.10 with the default peak tune [2][3]. Each benchmark is run ten times sequentially and the average of each metric is reported. The ref input is used for the SPEC benchmarks and the LDBC-1000k input is used for the GraphBig benchmarks. After the performing the experiments of one benchmark, the emitted profile is deleted and the system is restarted to perform the experiments of the next benchmark. For the SPEC benchmarks, only the first billion instructions (approximately) over all application threads are considered starting from the first instruction of the benchmark. For the GraphBig benchmarks, the first billion instructions starting from the parallelized region are profiled. That is sufficient because our goal here is not to analyze the behavior of the benchmarks themselves, but to measure the performance of Alleria itself using different configurations. Also note that these results were collected only in the process in which the respective benchmark was running. Any other processes (created by runspec) do not have any impact on these results.

Field	Value
Processor width	64-bit
Microarchitecture	Intel Broadwell
Number of cores	16 (No hyperthreading)
Frequency	1.70 GHz (Fixed)
Last-level cache	20 MB
Main Memory	32 GB DRAM DDR4
Hard drive sequential write throughput	170 MB/s
Hard drive random write throughput	1 MB/s
OS	64-bit Windows 10

Table 1: Description of the platform used to run the experiments.

Production operating systems such as Windows and Linux include sophisticated I/O schedulers that take advantage of the superior sequential I/O performance. In addition, they include a file system caching mechanism that services disk I/O requests from main memory (which has a much higher bandwidth) and only when it is deemed necessary, the requests will be performed on the disk drive. This enables the scheduler to find more opportunities to approach sequential I/O performance. We do take these feature into account. Profiling one billion instructions will quickly overwhelm the system cache.

5.2 General Results

[2]We had to make minor changes to the code of 403.gcc to get it compiled.

[3]We ported GraphBig CPU to Windows by making minor changes to the code.

The benchmarks used in this section include all the C and C++ SPEC CPU benchmarks and all the GraphBig CPU benchmarks. It is important to note that the performance of Alleria does not depend on the behavior of individual threads, but rather on the behavior of all of them collectively (despite the fact that multithreading makes the program run faster if run without profiling). Therefore, all results shown are averaged across all benchmarks and the average is rounded to the nearest integer. Alleria was configured to use four processing threads and a number of buffers equal to the number of app threads, each is 4 MB in size [4]. In the next subsection, we evaluate the impact of these parameters on performance. Figures 5 and Figure 6 show the execution time and profile size, respectively. We run Alleria with different configurations to determine how different features impact performance and the size of the generated profiles.

In Figure 5, the analysis configuration refers to enabling Alleria to analyze instructions and fill buffers but without actually writing them to the output profiles. Therefore, this configuration shows the overhead of Alleria excluding the buffered synchronous file I/O overhead. The Emit configuration refers to running Alleria normally with profiles generated. In all figures, nd indicates that only addresses and instructions are captured and no data is captured, d indicates that data is captured, ni indicates that no instructions are captured, txt indicates that the generated profile is textual, bin indicates a binary profile, and the adaptive configuration refers to using adaptive profiling and emitting binary profiles with data and instructions. Only the SPEC benchmarks were used for these configurations. The GraphBig benchmarks were used for the fixed N and adaptive N configurations in which N app threads were used and Alleria was configured as in emit d bin.

Generating textual profiles is extremely expensive. The processing threads have to do more computation to reconstruct the traces from the buffers and emit them into output profiles. This not only results in substantial execution overhead as shown in Figure 5, but also very low disk bandwidth utilization as shown in Figure 8. At the same time, the purpose of textual profiles is to manually inspect. This is only practical for a small number of instructions (few thousands or tens of thousands). Generally, binary profiles should be used. Binary profiles can be emitted more quickly and their sizes are much smaller.

For the SPEC CPU benchmarks and the GraphBig benchmarks with 4 app threads, adaptive profiling achieves the best performance over most statically specified number of processing threads and buffers. Figure 7 and Figure 8 show what the CPU utilization and disk write throughput look like when using adaptive profiling, respectively. The average CPU utilization and the average disk write through-

[4]For the multithreaded benchmarks, the parallelized region of each of them consists of multiple phases. At the end of each phase, there is a global barrier that prevents any app thread from crossing it until all app threads reach it. If some app threads reached the barrier but the others happened to be waiting for a free buffer, a deadlock will occur. We can ensure that this does not happen by having at least as many buffers as app threads. This is important too for single-threaded benchmarks, but only on Windows 10+ as explained in Section 5.3. We use the same number of processing threads for both SPEC and GraphBig benchmarks to allow comparison.

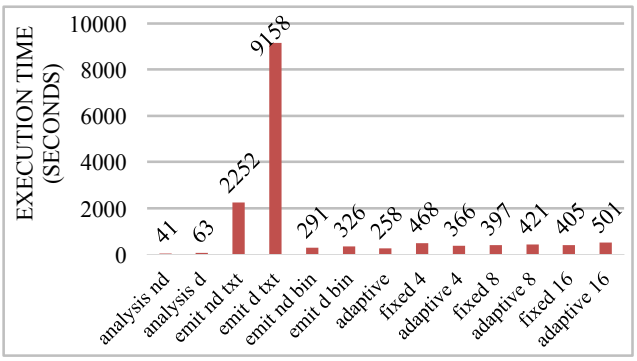

Figure 5: Execution time of Alleria with different features used.

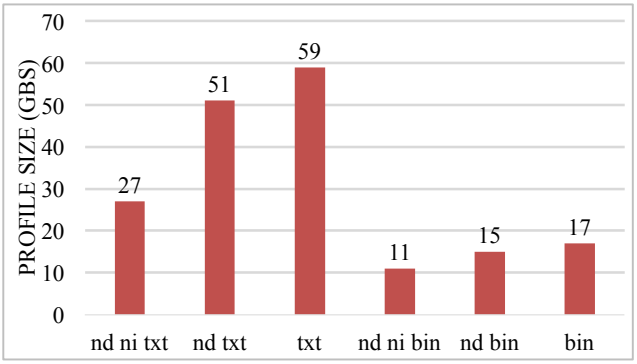

Figure 6: The sizes of profiles generated in different configurations without using any compression techniques.

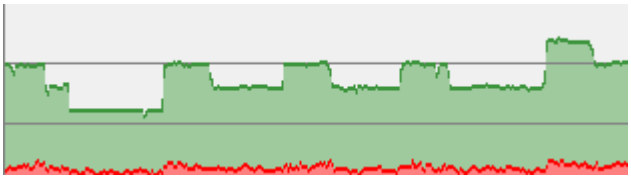

Figure 7: CPU utilization achieved using adaptive profiling for a duration of three minutes. Each horizontal line marks 15%.

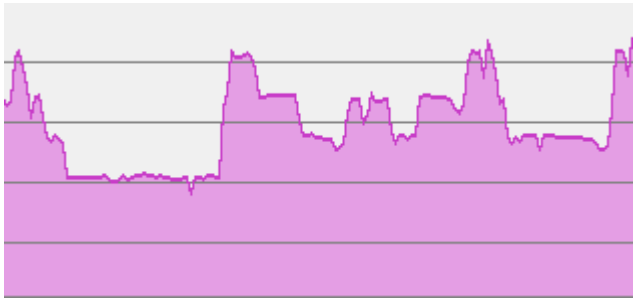

Figure 8: Disk write throughput achieved using adaptive profiling for a duration of three minutes. Each horizontal line marks 35 MB/s.

put are 28% and 119%, respectively. When CPU utilization decreases, this indicates that a processing thread was suspended, and when it increases, it indicates that a processing thread got resumed or a new one got created. Note how CPU utilization and disk write throughput both roughly rise and fall together, indicating that the adaptive mechanism is interacting very well with the OS file system caching mechanism. This nice pattern is generally impossible to achieve with static configuration as the processing threads will continuously overwhelm system cache or not utilize it fully. In the next subsection, we show the corresponding measurements for many different static configurations.

For the GraphBig benchmarks, adaptive profiling achieved better performance with a small number of app threads compared to static configuration. For 8 and 16 app threads, we observed that it tends to spend more time than necessary in a Z phase, resulting in inferior performance.

Figure 5 also shows that emitting profiles constitutes a larger overhead than instrumentation and data collection in buffers. Although the overhead of the latter is not negligible. Also we note that emitting the values being read or written does not increase overhead significantly. Figure 6 shows that those values occupy about 12% of resulting profile for the SPEC CPU benchmarks while dynamic instructions occupy about 34%. Therefore, it might be worth spending more effort to extract the instructions from the executable binaries rather than recording them with the profiles. On the other hand, once could use some of the trace compaction or compression techniques (discussed in Section 8.1) to reduce the space occupied by instructions (and other collected information).

We refer to executing a program without any profiling as native execution. The average native execution time for the benchmarks used in this subsection is roughly one second.

ptc	cpu util	disk bw	rt	total ptit	total ptpt	atwt	total npb	total ub
0	6	22	811	N/A	N/A	N/A	616228	11829630
2	13	49	447	132	768	336	636857	11865541
4	16	66	322	131	1250	278	616323	10937445
8	11	42	504	1815	2436	456	613715	10707202
16	9	31	634	7679	2635	577	654592	12469534
32	6	19	851	25323	3765	800	639752	12221770

bc	cpu util	disk bw	rt	total ptit	total ptpt	atwt	total npb	total ub
2	7	17	1121	18532	2102	1070	636670	10935600
3	10	25	716	3956	2051	662	636990	11738898
5	12	41	506	1887	2465	462	618001	10921778
7	14	56	402	940	2438	356	627994	11490094
15	15	64	344	282	2727	292	610081	10051348
30	14	63	387	293	3076	333	617924	10899416
100	14	56	408	700	3242	354	615385	10912560
500	13	49	464	151	3703	409	610808	10413615

bs	cpu util	disk bw	rt	total ptit	total ptpt	atwt	total npb	total ub
3	12	40	540	2544	2830	566	15744130	11820931
16	12	41	517	1944	2714	513	2462154	10462344
512	14	41	505	1930	2665	503	77295	11082893
4096	15	64	360	1466	1748	293	9384	10363696
65536	15	63	379	1344	1866	292	603	12019595
131072	15	44	441	1635	2254	442	316	12438306
262144	14	41	497	1724	2443	456	154	11288427

Figure 9: The performance of Alleria with respect to different numbers of processing threads, buffers, and buffer sizes.

If Alleria was injected in a process but without any profiling (a window never opens), the overhead is within 5%. Therefore, when using window configuration to selectively profile processes, threads, or pieces of code, those parts of the code that are not profiled will run almost at full speed.

5.3 Detailed Results

The behavior of the application being profiled is an important factor that affects Alleria's performance. However, that factor is beyond the control of Alleria as it has to profile what it is instructed to profile. There are three potential factors that can have a significant impact on performance and are under the control of Alleria. Those factors include the number of processing threads, the number of buffers, and the size of a buffer. In this section, we will study how these factors impact Alleria's performance using only the single-threaded bzip2 benchmark (other SPEC benchmarks yield the same conclusions). The configuration used includes emitting binary profiles with all features enabled (all instructions, virtual addresses, physical addresses, accessed values, thread scheduling information, and function names). The default values, 8 processing threads and 5 buffers each 64 KB in size, take effect when the corresponding factor is not being varied.

First, we need to define some acronyms: ptc stands for number of processing threads, cpu util stands for the average CPU utilization considering all the cores, disk bw stands for the average hard drive write throughput observed during profiling in MB/s, rt stands for execution time in seconds, total ptit stands for the sum of the time over all processing threads in which they remained idle in seconds (waiting for buffers to process), total ptpt stands for the sum of time over all processing threads in which they were processing buffers, atwt stands for the total time over all application threads during which they were waiting for free buffers in seconds, total npb stands for the number of processed buffers over all processing threads, ub stands for the amount of space in buffers that was not used summed over all processed buffers in KBs, bc stands for the total number of allocated buffers, and bs

stands for the size of a buffer in KBs (all buffers are of the same size). Note that ptc, bc, and bs can be specified by the user as command-line parameters.

The following figures are colored in a way to ease interpretation. The blue color is used to indicate that larger values are better. The red color indicates smaller values are better. Columns without colors are neutral or under study.

Figure 9 illustrates the impact that the number of processing threads has on the execution time of Alleria. The idea of processing threads is indeed useful. Using only two processing threads improves performance by 45%. This enables the application threads to spend much more of their time on execution application code in parallel to processing the buffers. Note that either way, application threads still have to run Pin's analysis routines (the code that captures the information to be collected). Overall, we can see the improvement of execution time is a reflection of the improvements in CPU utilization and disk write throughput.

However, as the number of processing threads increases, the benefit that they can achieve reduces and eventually, performance becomes worse than not using any processing threads. The reason for this is that the OS file system caching mechanism would find it more difficult to construct a sequential I/O schedule when there are many requests to many files. To effectively benefit from a large number of processing threads, multiple secondary storage devices need to be used. The ptit metric can be used as a clear indicator of when there are too many number of processing threads. It worth noting that each thread consumes by default 1 MB of the address space for its stack. Therefore, creating an extremely large number of threads can deplete the address space or cause serious fragmentation, which particularly problematic in 32-bit address spaces. Also note that a very large number of processing threads would significantly increase performance when the number of instructions to be profiled is small (less than 250 million) because of the effect of the system cache. In addition, we observed experimentally that gradually creating a large number of processing threads (e.g., 16) is better than creating all of them at once.

The number of buffers processed is different in different runs. This has little to do with the number of processing threads. Alleria does not exactly stop when a billion instructions are executed. Instead, it receives an asynchronous notification when that event occurs and then only stop profiling at a convenient time. Therefore, it typically executes a few more instructions than what is required, which can vary across different runs. The amount of unused space from buffers varies accordingly. The amount of unused space is about 17 KB per buffer of size 64 KB. That is, 25% of a buffer on average is not being used. This not an issue because the total number of buffers that is allocated is fixed (in this case, 5 buffers). The reason that there is wasted space is because Alleria instruments the application at the trace granularity and therefore, has to account for the maximum space demanded by that trace.

The number of buffers is another important factor as shown in Figure 9. When the number of buffers is too small, they become a bottleneck as both application threads and processing threads sit idle for long periods of time waiting for buffers to become available. As the number of buffers increase, similarly to the number of processing threads, the performance reaches a saturation point after which increasing the number of buffers adversely impact performance.

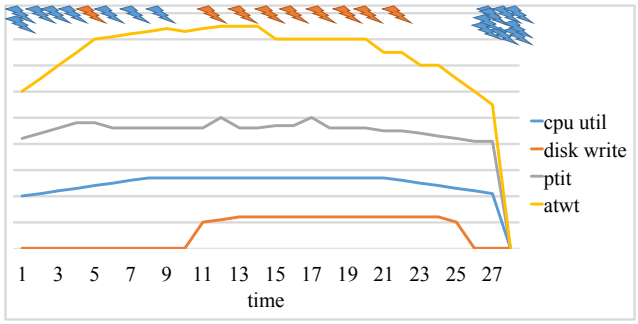

Figure 10: Adaptive profiling events plotted against application CPU utilization, application disk utilization, processing thread idle time, and application thread wait time. The blue and red lightning bolts at the top of the figure represent processing thread creation (or resumption) and suspension, respectively.

Running Alleria with a single buffer (bc = 1) is technically not possible on Windows 10 because the system uses parallel loading — there will be multiple application threads created by the OS for loading libraries and with a single buffer, a deadlock will occur. Although the performance resulting from using on buffer is likely to be worse than using two buffers.

The results in Figure 9 show that the size of a buffer may have a significant impact on performance, but not as much as the other two factors. A buffer size of few MBs to few dozen MBs yields the best performance. However, there are two caveats here. First, the window configuration manager works at a resolution determined by the buffer size. That is, it updates its data structures more frequently when the buffer size is smaller. Therefore, using very large buffers reduces the responsiveness of the window configuration manager. Second, the buffer size has to be large enough to accommodate the space requirements of any instruction trace. We observed that some traces require several KBs of memory when profiling some applications. For bzip2, 3 KB is the smallest size we could use.

In conclusion, the number of threads and the number of buffers both can be bottlenecks if they are too small. A very large number of processing threads substantially degrades performance when the number of instructions being profiled is large. The size of a buffer may constitute a bottleneck and only needs to be moderately large to successfully profile the target application.

5.4 Adaptive Profiling Illustration

The proposed adaptive profiling mechanism is not guaranteed to result in a performance that is superior to any that of the static configurations — it is a heuristic mechanism. We provide an illustration of how it works in this section. We developed a benchmark specifically design to behave radically differently over time to determine whether Alleria will be able to cope with it. The behavior of the benchmark is similar to that of the GraphBig benchmarks. We describe

how the benchmark works and how Alleria adapts to it.

The benchmark works as follows. A total of twelve thread are created serially, each is created every five seconds. The first eight threads each access a different memory location ten million times. The other four threads each write 50 KB to a different file on the hard drive. The total number of dynamic instructions is about 1.4 billion. Figure 10 shows how Alleria tries to adapt to the behavior of the benchmark. The figure shows how various important metrics change over time. That is, only changes are shown rather than absolute values. Also the CPU and disk utilization shown are only those resulting from the execution of the benchmark itself and not Alleria.

The proposed adaptive profiling mechanism is not guaranteed to result in a performance that is superior to any that of the static configurations — it is a heuristic mechanism. We provide an illustration of how it works in this section. We developed a benchmark specifically design to behave radically differently over time to determine whether Alleria will be able to cope with it. The behavior of the benchmark is similar to that of the GraphBig benchmarks. We describe how the benchmark works and how Alleria adapts to it.

The benchmark works as follows. A total of twelve thread are created serially, each is created every five seconds. The first eight threads each access a different memory location ten million times. The other four threads each write 50 KB to a different file on the hard drive. The total number of dynamic instructions is about 1.4 billion. Figure 14 shows how Alleria tries to adapt to the behavior of the benchmark. The figure shows how various important metrics change over time. That is, only changes are shown rather than absolute values. Also the CPU and disk utilization shown are only those resulting from the execution of the benchmark itself and not Alleria.

6. USE CASE: MEMORY ACCESS HEAT MAPS

We use a program that is similar to but slightly larger than the one shown in Listing 3 to generate memory access heat maps for the first 3 billion dynamic instructions of the 429.mcf benchmark. The benchmark was compiled with and without compiler optimizations, targeting 32-bit x86 in both cases [5]. In addition, we disabled ASLR so that the layout of the virtual address spaces in different runs are as close to each other as possible. Using the LLW, we observed that the base addresses of the call stack of the main thread, main executable, and libraries are exactly the same in all runs. However, there were minor variations in the locations of the memory regions that belong to the heap. All the runs were performed on the machine described in Table 1. Since Alleria is not guaranteed to exactly stop at 3 billion instructions, we manually ensured that the numbers of profiled dynamic instructions in all the runs are all within 1% of each other. This enables a fair comparison [6].

[5]The 32-bit Pin CRT has a bug that limits file sizes to 2 GB. Therefore, if the size of a profile surpassed that, it will round to zero and corrupt the profile. To circumvent this bug, we used a sufficiently large number of processing threads so that no profile exceeds 2 GB. This bug exists in all versions of Pin CRT.

[6]This can also be done by considering only the first 3 billion instructions from the emitted profiles. This is possible in single-threaded applications because Alleria timestamps all profiled instructions.

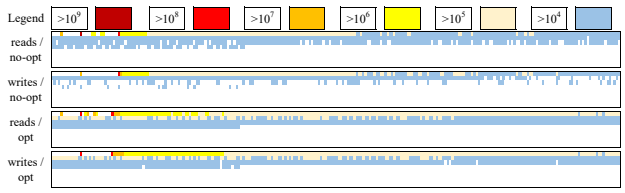

Figure 11: Heat maps illustrating the read and write memory access density for the 429.mcf benchmark without and with compiler optimizations.

The heat maps are shown in Figure 11. Each heat map consists of 8 rows and 256 columns. Each block represents a 1 MB chunk of the virtual address space. The block with the virtual base address of 0 is at the top left corner. Each row represents a 256 MB chunk of the address space and, therefore, each heat map represents the low 2 GB of the address space. The high 2 GB part is used only by kernel-mode code.

The first two heat maps show the read and write memory access density for the same run of an unoptimized version of 429.mcf, respectively. The other two heat maps show the read and write memory access density for the same run of an optimized version of 429.mcf, respectively.

The block with the most heat was accessed about 1.2 billion times. Without compiler optimizations, the total number of reads, writes, and accesses was 1.552 billion, 0.342 billion, and 1.894 billion, respectively. With optimizations, the total number of reads, writes, and accesses was 1.121 billion, 0.935 billion, and 2.056 billion, respectively. This implies that compiler optimizations may reduce the density of reads, but significantly increase the density of writes. The total number of accesses is also 5% denser (larger) than without optimizations. Moreover, Figure 11 shows that access heats are more distributed when optimizations are used.

These heat maps were produced assuming there is no memory hierarchy — there is simply a single level of memory. In a realistic system, there will be multiple levels of caches and different accesses will reach different levels of the hierarchy. Existing trace-driven tools can be easily integrated with the AbpHelp library to consume Alleria profiles.

7. RELATED WORK

We classify related work into three categories: replayers, execution-driven tools and trace-driven tools. The following three subsections discuss related works in each of these categories and compare them with Alleria. Some of these tools use a technique called memory shadowing [19, 18]. In this technique, every byte of memory has a corresponding byte (or some number of bits), known as the shadow byte, which says something about the history of values and the state of the address of that byte. This technique has two significant problems. First, it doubles the amount of physical memory used by the process. If this amount exceeds the threshold imposed by the OS, paging will occur causing severe performance degradation. Second, if the application being profiled

is multi-threaded, a reader-writer synchronization mechanism is required on every memory access to prevent shadow memory corruption. This also leads to performance issues. Such tools solve this problem by serializing the execution of the application. Alleria does not use memory shadowing.

7.1 Trace-Driven Simulation and Analysis Tools

METRIC [35] is similar to Alleria in that it generates memory access traces. METRIC's traces contain only the memory address of accesses and the addresses of the instructions that issued the accesses. METRIC uses a novel algorithm that detects regular access patterns to aggressively compress the generated traces. Such techniques substantially increase the profiling overhead.

Gleipnir [45] is another tool that generates memory access traces. It captures virtual addresses, access size, and associates accesses with variable names using Valgrind's support for parsing debug information. It is also capable of capturing physical addresses. The user can instruct Gleipnir to profile only certain sections of the code by inserting directives in the source code of the program to be profiled. Gleipnir can be configured to either profile a single process or the whole process tree. Alleria can be configured without changing the source code or having to recompile the code using window configuration and the user can select specific processes in the tree to be profiled. Gleipnir only emits textual profiles.

Dinero IV [46] is a trace-driven uniprocessor cache hierarchy simulator. It takes as input a trace file where each line contains a trace record. A trace record contains the access type, virtual address and size of a single memory access. Using this information, simulation is carried out according to a user-specified configuration. Because it does not depend on any profiling framework, its source code is OS-independent and ISA-independent. Dinero IV can be easily modified to use AbpHelp to simulate Alleria traces.

Adept [20] is a profiler based on the DynamoRIO instrumentation framework designed for the IA-32 architecture. It produces two types of binary profiles called DEPc and DEPm. DEPc compactly stores the addresses of basic blocks that have been executed so that one can recover the control flow of the target application. DEPm compactly stores the virtual address, size and type (read or write) of every memory access that has occurred during profiling. A number of techniques have been used to store profiling information compactly.

7.2 Execution-Driven Simulation and Analysis Tools

Cachegrind [22] is a cache hierarchy and branch predictor statistical simulation tool implemented using the Valgrind [47] instrumentation framework. It supports configurable L1 split data and instruction caches and LL unified cache. It uses virtual addresses and supports profiling child and forked processes (where each process writes to a separate profile). The generated profile includes statistics such as the total number of cache misses and the total number of memory references. These statistics can be viewed at different granularities including function level, line level and instruction level. Other tools are also provided to merge or compare profiles from different simulations.

CMP$im [48] is a cache hierarchy and coherence simulator implemented using the Pin framework. The user can specify in detail each level of the cache hierarchy to be simulated. The generated profile includes statistics such as the total number of cache accesses and misses, the sharing characteristics of multi-threaded applications and coherence traffic. It can also produce these statistics periodically to enable the user to understand how cache access behavior changes over time.

Memcheck [12] is a memory error detection tool implemented using the Valgrind framework. For each memory location in the user-mode virtual address space of the process, it defines a state that can be used to determine whether that location is addressable (that is, allocated by the application). Also for each allocated memory byte, it maintains a semi-bit-precise state that is used to determine whether a bit has a defined value or not (that is, whether the application has ever written a value to it). When the profiling session terminates, Memcheck reports all invalid memory accesses, use of undefined values, memory leaks and repeated frees of memory blocks. Memcheck is based on the shadow memory technique.

Dr. Memory [49] is another memory error detection tool implemented using the DynamoRIO framework [50]. It uses memory shadowing similarly to Memcheck. In particular, it uses the Umbra memory shadowing framework. However, its uninitialized memory access detection feature works at the byte granularity. Dr. Memory supports Windows, Linux and MacOS operating systems and runs only on the IA-32 architecture (even though DynamoRIO and Umbra support x86-64). In addition to reporting, traditional memory errors, Dr. Memory can also report (Windows only) handle leaks, GDI usage errors and invalid arguments passed heap APIs. It includes a tool called Strace that records a trace of all system calls executed by an application on Windows. Alleria can also be used for those purposes.

Memcheck and Dr. Memory use dynamic instrumentation and incur significant slowdowns. These tools are specialized and can achieve smaller overheads than Alleria. Google Sanitizers [51, 52, 53] use compile-time instrumentation to detect similar errors and incur much smaller slowdowns. However, the source code is required and needs to be recompiled to enable sanitizers. Alleria does not require the source code.

NTrace [54] is a function boundary tracing toolkit for the Windows operating system. It consists of a software driver for tracing kernel code, a DLL for tracing user-mode code and a program that injects the DLL into the target process. For every Windows function, NTrace overwrites the first 2-bytes of the function with a short jump instruction to a trampoline which in turn calls a thunk that performs the tracing. NTrace integrates with the Structured Exception Handling (SEH) infrastructure to capture abnormal exists from functions and stay in control. Alleria can be used to trace functions but only in user-mode.

PiPA [40] is one of the many profiling techniques that takes advantage of multi-core platforms by creating a number of threads that perform the required analysis on the generated traces. It is similar to what we have discussed in Section 2.1 except that it was designed for execution-driven fine-grained analysis or simulation. It also uses multiple

buffers following the producer-consumer model. It stores all the necessary information about every basic block that has executed to reconstruct the execution of that code. This includes control flow, the addresses, sizes and types of memory accesses. All of this information is stored in a compact binary format called Runtime Execution Profile (REP). The authors have implemented this technique using the DynamoRIO framework targeting the IA-32 architecture.

EDDI [55] is a debugging framework that uses dynamic instrumentation to implement low-overhead rich debugging features with particular focus on software watchpoints. For each memory page used by the application, EDDI allocates a shadow page containing shadow tags that specify which locations of application memory page are being watched. It also employs page protection attributes to efficiently watch any number of watchpoints within a range of memory addresses, reducing the instrumentation overhead. EDDI was implemented using DynamoRIO and targets the IA-32 and x86-64 architectures. It only extends an existing debugger and so does not produce any output.

Most of these tools can be more easily implemented as trace-driven based on Alleria as discussed in Section 1.

7.3 Replayers

PinPlay [56] is a framework for deterministically replaying the execution of parallel programs. It is based on the Pin DBI framework. Rather than recording the state of all registers and memory before and after every instruction, it first captures the initial state of the running program and then only records state changes caused by non-deterministic events such as system calls, order of accesses to shared memory and uninitialized memory location reads. As opposed to the approach used in Alleria, to capture the memory state during execution, PinPlay uses the shadow memory technique. This doubles the memory requirements of the application being instrumented. PinPlay emits all captured information in the pinball format [57]. A pinball is similar to an Alleria binary profile in that it is portable and highly compressible. Replayers have an advantage is that the profiles they generate tend to be significantly smaller since only the information that is necessary to replay the execution of the app is recorded while everything that has been computed will be recomputed during replay. Deterministic replay is mostly useful for debugging data races. Alleria does not support deterministic replay. On the other hand, the overhead of running the program must incurred every time a pinball is used by an analysis or simulation tool.

8. DISCUSSION

Alleria is a profiling framework that generates detailed instruction and memory traces. Those traces can then be fed into trace-driven tools to potentially process the traces and simulate one or more target platforms. There are several advantages to trace-driven simulation. First, the input program only has to be executed once to generate a trace and then simulation can be performed many times. Second, trace-driven tools are much easier to design and implement. Third, trace-driven tools can achieve the fastest simulation throughput compared to other approaches. Fourth, simulation can be performed on a machine that is different from and potentially incompatible with the input program.

That said, trace-driven simulation has also disadvantages. First, trace files are typically very large in size. Therefore, emitting these files adds a significant overhead to profiling, storing and copying these files can be inconvenient for users, and consuming these files adds an overhead to trace-driven tools.

Even though a single trace file may be used to perform many simulations for different target platforms, this may not accurately capture the behavior of the input program on all or any of these platforms. This brings to the second disadvantage. Generating multiple trace files may be necessary. That reason for this is that the program may not behave deterministically every time it runs [58]. This can happen due to many reasons including OS thread scheduling, OS I/O scheduling, inter-thread communication and synchronization, hardware prediction algorithms, memory management algorithms, and any other nondeterministic mechanisms in the host platform, target platform, and the input program.

At the same time, generating a single trace file might be sufficient despite any nondeterministic aspects of execution if the trace-driven tool to be used is designed to be aware of these issues and take them into account.

8.1 Techniques to Reduce Profiles Sizes

Memory traces tend to be very large in size. Even if a window configuration was used to profile only specific functions or parts of a program, it is likely that these function contain iterative control flow with large trip counts. The addresses of the locations being accessed across iterations are usually related. Rather than storing information about every memory access in the profile, once could record the base addresses of the data structures being iterated over and an algorithm or a function that specifies how to generate the rest of the accesses. This approach has been studied extensively [59, 60, 35]. It is very effective in compacting the emitted memory traces without sacrificing fidelity, particularly when the application being profiled spends most of its time in executing loops. We did not use here any of the previously proposed techniques, although most of them are applicable.

9. CONCLUSION AND FUTURE WORK

We developed a new framework for generating instruction and detailed memory traces. A novel configuration enables the user to easily select the instructions of interest to be profiled. Even though Alleria has a large overhead, window configuration can help reduce this overhead by focusing the profiling on specific parts of the program. Interceptors is another convenient feature that can significantly reduce profiling overhead, but potentially making the trace-driven tool more complicated.

There are several improvements that can be made. The first one is to make Alleria intelligent in the sense that it can figure out by itself how many processing threads and buffers are required and dynamically adjust these parameters. The second improvement is using asynchronous file I/O while at the same time increasing the number of buffers. The third improvement is integrating Alleria with more debugging information. In particular, enabling the user to map memory addresses to local and global variables. We might also con-

sider enabling the user to specify different file paths for different processing threads' profiles. Ideally, the paths would be in different storage devices. This might reduce the overhead. Some of the techniques to reduce profile sizes can be incorporated. Window configuration can be enhanced to support sampling [61].

Interceptors can be made potentially more useful by translating the binary code to some intermediate language and providing built-in support for it. Currently, arguments and return values can only be captured and not modified. The ability of modifying the arguments or return values of functions can be useful for finding vulnerabilities using fuzzing.

10. REFERENCES

[1] M. Marinella, "The future of memory," in *Aerospace Conference, 2013 IEEE*, pp. 1–11, IEEE, 2013.

[2] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Memory Workshop (IMW), 2013 5th IEEE International*, pp. 21–25, IEEE, 2013.

[3] P. Tschirhart, J. Stevens, Z. Chishti, S.-L. Lu, and B. Jacob, "Bringing modern hierarchical memory systems into focus: A study of architecture and workload factors on system performance," in *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, (New York, NY, USA), pp. 179–190, ACM, 2015.

[4] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[5] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte, "Energy efficient phase change memory based main memory for future high performance systems," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8, IEEE, 2011.

[6] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[7] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 2–13, ACM, 2009.

[8] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 14–23, ACM, 2009.

[9] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun, "Sorting with asymmetric read and write costs," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, (New York, NY, USA), pp. 1–12, ACM, 2015.

[10] S. Wen, M. Chabbi, and X. Liu, "Redspy: Exploring value locality in software," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 47–61, ACM, 2017.

[11] L. Song and S. Lu, "Performance diagnosis for inefficient loops," in *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, (Piscataway, NJ, USA), pp. 370–380, IEEE Press, 2017.

[12] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision.," in *USENIX Annual Technical Conference, General Track*, pp. 17–30, 2005.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[14] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach, "Dynamic program analysis of microsoft windows applications," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 2–12, IEEE, 2010.

[15] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12, IEEE, 2011.

[16] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[17] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones, "A trace-driven approach for fast and accurate simulation of manycore architectures," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pp. 707–712, IEEE, 2015.

[18] Q. Zhao, D. Bruening, and S. Amarasinghe, "Efficient memory shadowing for 64-bit architectures," in *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, (New York, NY, USA), pp. 93–102, ACM, 2010.

[19] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 22–31, ACM, 2010.

[20] Q. Zhao, J. E. Sim, W.-F. Wong, and L. Rudolph, "Dep: Detailed execution profile," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, (New York, NY, USA), pp. 154–163, ACM, 2006.

[21] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pp. 28–36, 2008.

[22] N. Nethercote, "Dynamic binary analysis and instrumentation," tech. rep., University of Cambridge, Computer Laboratory, 2004.

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[24] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, pp. 14:1–14:25, June 2017.

[25] X. Dong, C. Xu, N. Jouppi, and Y. Xie, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, pp. 15–50, Springer, 2014.

[26] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[27] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[28] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 113–127, ACM, 2016.

[29] M. Russinowich and B. Cogswell, "VMMap," 2015.

[30] M. Russinowich and Bryce, "RAMMap," 2016.

[31] V. M. Weaver and S. A. McKee, "Are cycle accurate simulations a waste of time," in *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, pp. 40–53, 2008.

[32] O. Golovanevsky, A. Dayan, A. Zaks, and D. Edelsohn, "Trace-based data layout optimizations for multi-core processors.," in *HiPEAC*, pp. 81–95, Springer, 2010.

[33] C. Meng, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Efficient memory

partitioning for parallel data access in multidimensional arrays," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2015.

[34] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A new approach to automatic memory banking using trace-based address mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 179–188, ACM, 2017.

[35] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo, "Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies," *ACM Trans. Program. Lang. Syst.*, vol. 29, Apr. 2007.

[36] E. Park, C. Kartsaklis, T. Janjusic, and J. Cavazos, "Trace-driven memory access pattern recognition in computational kernels," in *Proceedings of the Second Workshop on Optimizing Stencil Computations*, WOSC '14, (New York, NY, USA), pp. 25–32, ACM, 2014.

[37] D. L. Schuff, B. S. Parsons, and V. S. Pai, "Multicore-aware reuse distance analysis," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, IEEE, 2010.

[38] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, (New York, NY, USA), pp. 191–196, ACM, 2012.

[39] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[40] Q. Zhao, I. Cutcutache, and W.-F. Wong, "Pipa: Pipelined profiling and analysis on multicore systems," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 13:1–13:29, Dec. 2010.

[41] W. Zhang, B. Calder, and D. M. Tullsen, "An event-driven multithreaded dynamic optimization framework," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 87–98, IEEE, 2005.

[42] D. Upton, K. Hazelwood, R. Cohn, and G. Lueck, "Improving instrumentation speed via buffering," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, (New York, NY, USA), pp. 52–61, ACM, 2009.

[43] T. S. P. E. C. (SPEC), "The standard performance evaluation corporation (spec)," 2006.

[44] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pp. 1–12, IEEE, 2015.

[45] T. Janjusic and K. Kavi, "Gleipnir: A memory profiling and tracing tool," *SIGARCH Comput. Archit. News*, vol. 41, pp. 8–12, Dec. 2013.

[46] J. Edler, "Dinero iv trace-driven uniprocessor cache simulator," *http://www. cs. wisc. edu/˜ markhill/DineroIV/*, 1998.

[47] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[48] A. Jaleel, R. S. Cohn, C.-k. Luk, and B. Jacob, "Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps," *tech. rep., UMD-SCA*, 2006.

[49] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (Washington, DC, USA), pp. 213–223, IEEE Computer Society, 2011.

[50] D. Bruening and S. Amarasinghe, *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[51] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker.," in *USENIX Annual Technical Conference*, pp. 309–318, 2012.

[52] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pp. 46–55, IEEE, 2015.

[53] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with llvm compiler," in *International Conference on Runtime Verification*, pp. 110–114, Springer, 2011.

[54] J. Passing, A. Schmidt, M. von Lowis, and A. Polze, "Ntrace: Function boundary tracing for windows on ia-32," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 43–52, IEEE, 2009.

[55] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "How to do a million watchpoints: Efficient debugging using dynamic instrumentation," in *International Conference on Compiler Construction*, pp. 147–162, Springer, 2008.

[56] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, (New York, NY, USA), pp. 2–11, ACM, 2010.

[57] H. Patil and T. E. Carlson, "Pinballs: portable and shareable user-level checkpoints for reproducible analysis and simulation," in *Proceedings of the Workshop on Reproducible Research Methodologies (REPRODUCE)*, 2014.

[58] S. Nilakantan, S. Lerner, M. Hempstead, and B. Taskin, "Can you trust your memory trace? a comparison of memory traces from binary instrumentation and simulation," in *VLSI Design (VLSID), 2015 28th International Conference on*, pp. 135–140, IEEE, 2015.

[59] A. Hroub, M. E. S. Elrabaa, M. F. Mudawar, and A. Khayyat, "Efficient generation of compact execution traces for multicore architectural simulations," *ACM Trans. Archit. Code Optim.*, vol. 14, pp. 27:1–27:25, Aug. 2017.

[60] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "Loopprof: Dynamic techniques for loop detection and profiling," in *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

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