COMPUTER PROGRAM INSTRUMENTATION USING RESERVOIR SAMPLING & PIN++

A Thesis

Submitted to the Faculty

of

Purdue University

by

Brandon E. Upp

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2019

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. James H. Hill, Chair

Department of Computer and Information Science

Dr. Rajeev R. Raje

Department of Computer and Information Science

Dr. Mihran Tuceryan

Department of Computer and Information Science

Approved by:

Dr. Shiaofen Fang

Head of the Graduate Program

To my loving wife, Amanda. I could not have done this without you.

ACKNOWLEDGMENTS

I could not have done this alone.

Thank you to Dr. Hill, for your support and guidance.

Thank you to Dr. Raje and Dr. Tuceryan for serving on my committee.

Thank you to the support staff of the Computer Science department office for help with my many, many questions.

Thank you to the School of Computer and Information science for providing me with the funding and opportunity to perform this research.

Thank you to my wife Amanda for continued, patient support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Upp, Brandon E. MS, Purdue University, August 2019. Computer Program Instrumentation Using Reservoir Sampling & Pin++. Major Professor: James H. Hill.

This thesis investigates techniques for improving real-time software instrumentation techniques of software systems. In particular, this thesis investigates two aspects of this real-time software instrumentation. First, this thesis investigates techniques for achieving different levels of visibility (*i.e.*, ensuring all parts of a system are represented, or visible, in final results) into a software system without compromising software system performance. Secondly, this thesis investigates how using multi-core computing can be used to further reduce instrumentation overhead. The results of this research show that reservoir sampling can be used to reduce instrumentation overhead. Reservoir sampling at a rate of 20%, combined with parallelized disk I/O, added 34.1% additional overhead on a four-core machine, and only 9.9% additional overhead on a sixty-four core machine while also providing the desired system visibility. Additionally, this work can be used to further improve the performance of real-time distributed software instrumentation.

1 INTRODUCTION

To design, build, and maintain software systems, it is vitally important to be able to understand what happens inside of a program while it is running. Software instrumentation can provide insight into the inner workings of software systems. At a very basic level, software instrumentation is adding additional code to a program to gather information about the program itself [1]. There are many uses for instrumentation, such as debugging [2–4], system profiling [5–7], data flow analysis [8], and others. Tasks such as data flow analysis could also be performed by analyzing source code, but given the size and complexity of software system it can be simpler to instrument the program and have the system do the analysis itself [8].

There are many techniques for instrumenting programs. Two of these techniques are static instrumentation, and dynamic binary instrumentation. Static instrumentation [9] is adding instrumentation to a program before it is run. The source code is altered, and the program is recompiled and restarted as needed. A simple example would be adding print statements to a program to check the value of specific variables for debugging purposes. Dynamic binary instrumentation [10], or DBI, happens during runtime. DBI can be applied to programs as they start execution, or attached to running processes. Instrumentation code is inserted into the binary on-the-fly, without making any permanent changes. DBI analyzes the binary to determine where code should be inserted, and what code to insert. Unlike static instrumentation, dynamic binary instrumentation does not require recompilation of the instrumented program. Also, DBI will have access to both user code and system calls, as well as any self-generating or self-altering code [9, 10]. Some examples of of dynamic binary instrumentation frameworks are Pin [11], DynamoRIO [12], and Valgrind [13].

Instrumentation does not come without a cost, however. Adding more instructions to a binary means the program has more to do, and will take longer to run. The

amount of overhead will depend on the nature of the instrumentation. For example, analysis of every single instruction executed by a program will cause a great deal of overhead. The amount of extra overhead is mostly determined by the code added by the user [10]. This overhead could be anywhere from thirty percent, to many hundreds of percent more than the runtime of the original, uninstrumented program [10, 14]. Aside from increasing program runtime, instrumentation overhead can interfere with the instrumentation itself. For example, instrumentation for profiling can affect system performance, leading to inaccurate profiling results [7]. It is important to keep instrumentation overhead to a minimum.

Since much of the additional overhead of binary instrumentation is caused by user code, care must be taken when creating new instrumentation tools. As with any program, consideration must be given to code performance and memory use. However, there are additional techniques that may reduce instrumentation overhead. One such technique is using sampling. Instead of recording every possible result from instrumentation, a sample of the results could be taken so that only a portion of the results are recorded. Recording fewer results means less work, which would decrease overhead. A second possible technique is to use parallelization when designing the instrumentation tool. If some of the instrumentation work can be successfully parallelized, the work that comes from additional overhead could be spread out over multiple CPU cores, potentially decreasing overall runtime. One aspect of instrumentation that is a candidate for parallelization is recording the results. A separate component of the instrumentation tool could be responsible for writing results to disk, or even streaming them to a remote system, while the main part of the tool continues to analyze the underlying program.

With this understanding, the contributions of this thesis are as follows:

- We have created an instrumentation tool that combines dynamic binary instrumentation with sampling and parallelization. This tool tracks the return addresses of function calls. The goal is to sample twenty percent of these addresses. There are two benefits to the design of our tool. First, it is relatively

simple to build a tool to track function call return addresses. Thus, more time could be spent on implementing the sampling and parallelization parts of this tool. Second, instrumenting every function call in a system will add a large amount of overhead, allowing us to see how our approach performs under stress.

- Our approach is novel because it takes into consideration system visibility. System visibility is how much of the inner workings of a system is known, or "visible." Our approach records memory addresses, and we want to sample twenty percent of these addresses. But this does not mean simply sampling twenty percent of all addresses. Instead, we want to ensure that we sample twenty percent of the return address of every function that is called in the underlying program.

- Our approach uses Pin and the framework Pin++ [15] to combine two ideas: Vitter's reservoir sampling algorithm [16], and a threaded component that attempts to parallelize disk I/O. Normally, a Pintool is single-threaded by default. When the tool needs to write data to disk the entire Pintool, as well as the underlying program, come to a halt until I/O is finished. An additional Reader component and a double buffer was added to the tool to allow for simultaneous reading and writing of sample data.

- By combining Reservoir sampling and parallelized disk I/O, we created a tool that is able to provide a representative data sample that achieves our desired system visibility, and increases overhead by an average of only 9.9%.

The results of this thesis that reservoir sampling was able to significantly reduce instrumentation overhead while still providing twenty percent system visibility. On a four-core system, exhaustively sampling every memory address increased runtime by an average of 85%. Multi-core reservoir sampling increased runtime by an average of only 34.09%. Added runtime was reduced by 59.9%. On a sixty-four-core system, exhaustive sampling increased runtime by an average of 69.6%. Multi-core reservoir sampling increased runtime by only 9.9%, an 85.8% improvement.

The Reader component of the multi-core Pintool was examined to see how much time it spent reading versus how much time it spent idle. The Reader spent less than 14% of its time reading, suggesting that the Reader could perform higher levels of sampling without adversely affecting performance. The multi-core version was tested at higher levels of sampling. The increase in runtime was much smaller than the increase in runtimes of other sampling methods tested at higher sampling rates.

1.1 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 briefly discusses other literature covering attempts to incorporate sampling and parallelization into instrumentation. Chapter 3 discusses Pin and Pin++, as well as Vitter's reservoir sampling algorithm. Chapter 4 describes how our tool was designed and created, as well as the challenge we faced in doing so. Chapter 5 describes the setup and design of our experiment as well as our experimental results. Finally, Chapter 6 provides concluding remarks and future research directions.

2 RELATED WORKS

This chapter is a brief review of techniques others have created to try and reduce instrumentation overhead. In particular, we compare our work to the areas of sampling and parallelization. Sampling improves performance by reducing the amount of data collected and thus reducing the amount of work done. Parallelization improves performance by spreading work over multiple CPU cores.

2.1 Sampling

One way of performing sampling is code duplication [17]. With this method, the code is duplicated and the duplicated code is instrumented. Most of the time the original code is executed, until a trigger causes a shift into the duplicated code so instrumentation can happen. Instrumentation does not happen often enough to cause significant overhead, but still happens often enough to provide useful results. This method is capable of creating accurate profiles with low overhead.

Bursty Tracing [18] extends the code duplication technique. The "bursts" of time spent in the instrumented duplicate code are increased, allowing this method to capture information the previous method could not. For example, bursts now can possibly cross over procedure boundaries, allowing for the collection of inter-procedure information. Additionally, Bursty Tracing was implemented using C and C++ on x86 processor. The previous method was limited to Java.

2.2 Parallelization

With multi-core CPUs being common nowadays, parallelization is a promising way to improve instrumentation performance. One method, SuperPin [19] splits an

instrumented binary into small, non-overlapping slices. The slices are run on separate threads in parallel, instead of sequentially like they would normally be. This allows for faster performance of the instrumented program. However, this is a difficult technique to implement, due to the complexity of ensuring slices are truly even and non-overlapping. The difficulty increases if the program is multi-threaded.

Another use of parallelization is *Pipelined Profiling and Analysis*, or PiPA [20]. Instead of splitting up the instrumented program, multiple threads are used as stages of a profiling pipeline. Small profiles are created on the same thread as the instrumented program, causing minimal impact on the program itself. These little profiles are then passed on the the profile pipeline threads, which assemble them into the complete profile and perform analysis.

A method called shadow profiling [5] uses both sampling and parallelization. The method is similar to code duplication and Bursty Tracing methods mentioned earlier. Shadow profiling also creates copies of the original application, called shadow applications. The shadow applications will contain the instrumentation code. However, instead of switching back and forth between un-instrumented and instrumented code, both the main application and the shadow applications are run in parallel. A monitor manages the creation and execution of shadow applications so as to collect enough sample to give good results without adding too much overhead.

None of these techniques work quite the same way as our approach does. We want our approach to give us twenty percent system visibility of all function calls, and to ensure that all functions are represented in the final sample. The issue with the methods that go back and forth between instrumenting and not instrumenting is that there is no way to sample an event that happens when you are not sampling [20]. PiPA records one hundred percent of events, so we know we wouldn't miss anything. However, our tool records the return addresses of function calls and is meant for long-running applications. Such a tool could potentially generate hundreds of megabytes of data a minute. Some kind of option for sampling is needed to keep the flow of data

to a manageable amount. Our approach is novel because it can provide the system visibility we want while still using sampling.

3 BACKGROUND

This chapter describes the tools used in implementing our approach. Pin is a framework for creating instrumentation tools called "Pintools." Pin was used in conjunction with Pin++, a framework that aids in creating Pintools. The sampling methods we used are discussed, including Vitter's reservoir sampling algorithm.

3.1 Pin

Our tool was created with Intel's Pin [21], a framework for dynamic binary instrumentation. Tools created with Pin, called Pintools, are written in C or C++.Code from the Pintool is injected into a program's binary at runtime. The Pintool specifies where in the binary analysis code should be injected. Code injection can happen as a program starts, or a Pintool can be attached to a running process.

Pin's method of injecting code into the binary provides a level of flexibility that static instrumentation methods lack. Since the code is dynamically injected into the program binary, access to the program's original source code is not needed. There is no need to re-compile the original program. Additionally, a Pintool is not limited to instrumenting a single program. Pintools can generally instrument any given program. Pin's API allows access to data that may be difficult, or impossible, for a program to access by itself. For example, as mentioned before, Pin can instrument code not originally present in the binary. This includes code from system libraries, self-generating code, and self-modifying code.

There are two main components to instrumentation. First, there needs to be a way to determine what parts of a program should be instrumented. Pin will inject analysis code, but it needs to know where the code should go. Second, the injected code needs to do something. Once the instrumented code reaches an instrumented

part, what will happen? We will want to do something that will provide us with information about the program. For example, consider the process of debugging a program. A simple and helpful way of doing this is adding print statements to the program to trace program execution. This is a very simple form of instrumentation. The "where" part of this instrumentation would be at the beginning of each function. The "what" part would be a print statement that outputs the function's name. This approach can be helpful, and simple to implement. However, adding print statements to a program requires access to the program's original source code. If the source is available this would not be a problem, but this is not always an option with closed source software. If the program is written in a compiled language, the program will have to be recompiled after the print statements are added, or removed. If the scope of instrumentation is big enough, the process will quickly become tedious and error prone.

With Pin, there is no need manually find instrumentation points and insert code by hand. When a program is run with a Pintool, the tool analyzes the binary to find the appropriate injection points. Callbacks are injected at these points. When the program reaches a point with a Callback, an analysis function is called. This analysis function contains the Pintool author's code. Continuing with the code trace example from above, Pin would analyze the program's binary and insert a callback at each routine it finds. This callback would call its corresponding analysis function, passing the name of the function as an argument. The analysis function would print the name of the function to the console, or to an external file, or do whatever else the author would like. Pin automates the process of inserting the code at all the correct points, and instruments the program without permanently altering the binary or original source code. Adding instrumentation to a program is as simple as running the program with the Pintool. To run the program without instrumentation, simply run the program without Pin. This is a very simple example. Pin is capable of instrumenting binaries at different granularities, from the higher level of a binary image, down to routines, or even individual instructions [22]. Multiple different callbacks and any

number of analysis routines can be used inside a single Pintool. A Pintool will have much more flexibility, and access to more information, than a simple print statement can.

The data gained from instrumentation can be valuable, but instrumentation always comes with a cost. The injected code adds extra runtime overhead, and instrumented applications run more slowly than their un-instrumented counterparts. More code means more work for the application. Even the simple example using print statements will add additional overhead. An extra print statement here or there may not be noticeable, but through instrumentation of a program can cause hundreds of thousands, if not millions of extra print statements a minute. That kind of overhead will be noticeable. It is wise to keep this extra overhead in mind, and strive to keep it to a minimum. For example, carefully instrumenting only the necessary parts of the program will reduce overhead. However, no amount of careful Pintool design can completely eliminate overhead. As far as the runtime of the original program is concerned, the Pintool's analysis routines are entirely overhead. Every line of code in the Pintool will slow the original program down.

If instrumentation code will always add overhead, we can consider ways to have the code run less often. One way of accomplishing this is to reduce the amount of data recorded by using sampling. If only a portion of available data is recorded, then the amount of work, and thus the amount of added overhead, is reduced. Careful sampling of the data provided by instrumentation can reduce overhead while still providing useful results.

3.2 Sampling Methods

Instead of collecting every piece of data from an instrumentation tool, we can instead sample a smaller portion of that data. Collecting less data means less additional overhead is added to the instrumented program. Consider a tool that samples an arbitrary percentage of data. Say, fifty percent of all function calls.

Each function call is an "event" that has a fifty percent chance of being recorded. There are several ways that such events can be sampled. Our experiments used four: percentage-based sampling, constant sampling, reservoir sampling, and exhaustive sampling.

3.2.1 Percentage-based Sampling

With percentage-based sampling, each time an eligible event occurs, a percentage is calculated to determine if the event is recorded. This is a simple method for randomly sampling events. However, a downside of this method is the constant need to calculate a percentage for every single event. Even though we only want to sample twenty percent of all events, a percentage calculation happens for one hundred percent of events. These constant calculations add considerable overhead.

3.2.2 Constant Sampling

Another possibility would be constant sampling. Instead of using a percentage, simply sample every nth element. In the case of twenty percent sampling, every other event would be recorded. Instead of constantly calculating percentages, a simple counter would keep track of whether or not an event is sampled. This removes some of the overhead caused by percentage sampling.

3.3 Reservoir Sampling

A third type of sampling is Vitter's reservoir sampling method. Vitter's reservoir sampling algorithm is a computationally-efficient method of sampling that selects a random sample of n records from a total data set of N records, where the size of N is not known beforehand. This method calculates a statistical distribution that ensures that each of the N total records have the same probability of ending up in the final reservoir of n records, even though the final size of N is unknown beforehand. An

important aspect of Vitter's reservoir sampling algorithm is that it is a single-pass algorithm. There is no need to first count the total number events to determine how many events should be sampled. The algorithm tracks the total number of events, as well as the number of events already sampled. It uses this information to calculate a probability distribution such that every event has an equal chance to be selected, even though the total number of events is unknown beforehand. The algorithm works in three stages.

In the first stage, a reservoir size of n is chosen. The size of the reservoir is the number of samples that will wind up in the final results. During the initial stage, every event is sampled until the reservoir has been filled. Next begins the second stage. At this point, the algorithm begins calculating how many events to simply skip over before before taking another sample. This number of events to skip over is called the "skip constant". This calculation is how the algorithm ensures an even probability distribution. One the appropriate number of events have been skipped, the chosen sample then randomly replaces one of the elements already in the reservoir and a new skip constant is calculated.

The second stages runs until a certain number of events have occurred. After this threshold is reached, the algorithm enters its third stage which uses a different method for calculating how many events to skip. This method was found to be more efficient once a large-enough amount of data had already been processed. The algorithm keeps sampling calculations to a minimum to reduce the amount of additional overhead.

3.4 Pin++

Our Pintool was not written directly with Pin. Instead, the tool was written using Pin++ [15], an object-oriented framework. Pin++ uses aspects of C++, "such as abstraction, encapsulation, and template metaprogramming" to reduce Pintool complexity, and increase Pintool performance. The authors found that using Pin++ lead to up to a 54% decrease in Pintool complexity and up to 64% less instrumentation

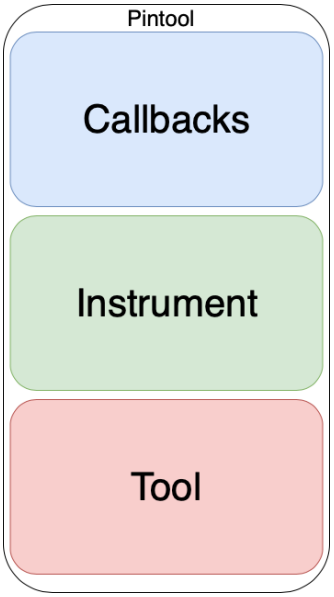

Figure 3.1.: Basic structure of a Pintool in Pin++

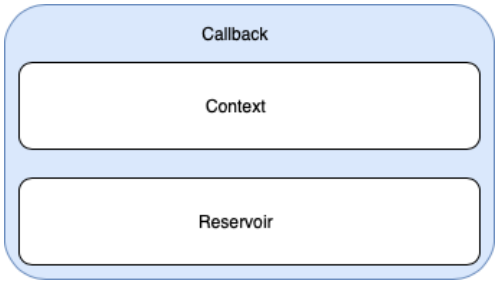

Figure 3.2.: Structure of the Callback used in our tool. Each function will have its own callback

overhead. They also found that with Pin++ there was greater modularity in Pintool components, allowing for reuse in other Pintools.

As shown in Figure 3.1, Pintools written in Pin++ have three main parts:

- Tool. The connects Pin with the rest of the Pintool. This usage of the upper-case word "Tool" should not be mistaken for the more general lower-case "tool," which broadly refers to all instrumentation tools. The Tool is responsible setting the Pintool up when the instrumented program starts, and finishing up when the instrumented program ends.

- Instrument. The instrument is responsible for determining at which points the target program is instrumented, and what callbacks to insert at those points. While our Pintool only has one Instrument, Pintools can contain an unlimited number of Instruments.

- Callback. This is called when the corresponding instrumentation point in the application is executed. The Callback can contain whatever analysis code the Pintool author wishes to use.

Figure 3.2 shows the structure of the Callbacks used in our tool.

4 RESERVOIR SAMPLING PINTOOL IMPLEMENTATION

This chapter will discuss the implementation of our approach, which was built upon previous work. First is a discussion of some of the challenges in implementing our approach. Next are details about the makeup of the reservoir sampling tool. Finally, the addition of a Reader that attempts to create a multi-core tool that parallelizes disk I/O.

4.1 Implementation Challenges

The first challenge we faced was ensuring system visibility. Instead of simply sampling a percentage of all function return addresses in a system, we wanted to ensure that every function was sampled at the same percentage. This way there was less risk of commonly-called functions being oversampled, rarely-called function being undersampled, and we could ensure that every function would be present, or "visible," in the final output.

All functions should be represented in the final output. Some functions will be called often, and be represented many times in the final sample. Other functions will be called much less often, or possibly only once or twice during the life of the program. Constant sampling would have the lowest added overhead, but there will be a risk of some functions not appearing in the final sample.

Percentage-based sampling and reservoir sampling can both prevent this issue of events falling in-between samples, but neither can guarantee that a given function is ever sampled. Because reservoir sampling is more computationally efficient than percentage-based sampling, we chose to work with reservoir sampling.

To solve the problem of system visibility, reservoir sampling is separately applied to each function in the program. Each function has its own reservoir, and the aim is

to sample the same percentage of each function. Functions that are called more often will have larger reservoirs, and functions called less often will have smaller reservoirs. The state of the reservoir sampling algorithm is tracked individually for each function. When a function is considered for sampling, only that function's state is used in the calculations. In this way, we can ensure that each function is sampled the correct amount, and we achieve the desired system visibility.

The next issue is with the way reservoir sampling works. Reservoir sampling is meant for a finite data set of unknown size. How many events will be sampled is known in advance, and determines the size of the reservoir. The algorithm runs, and the end result is the desired number of samples. Our instrumentation tool is meant to sample a percentage events from a long-running process. Reservoir sampling requires a specific number for the size of the reservoir, not just a desired sampling percentage. Without knowing how large the data set is in advance, there is no way to know what a given percent of the total number of events will be. Additionally, since the final number of sampled events is unknown there is a risk that the number of samples will grow so large as to not be able to fit in memory. The longer the process runs, the more likely it is that memory will run out.

Our approach uses the reservoir sampling algorithm in such a way as to avoid this problem. The algorithm itself is unchanged. However, instead of sampling the entire runtime of the instrumented program as one, long stream of events, the sampling is broken up into user-defined time intervals. At the beginning of each interval, the algorithm begins anew from stage one with an empty reservoir. The algorithm continues normally until the interval is over. At the end of each interval, the current contents of the reservoirs are flushed. The total number of events from the previous interval is used to calculate the size of the reservoir for the next interval. Reservoirs also have a minimum size. This way, even if a routine is not called during a time interval, it will still have a reservoir in case it is called during the next interval. This prevents issues of small, or even zero-sized reservoirs. After everything is reset,

the algorithm restarts with an empty reservoir. Reservoir sizes are adjusted as the program runs to try and achieve the desired sampling percentage.

Generally, Pintools are single-threaded. When one part of the Pintool is executing, the rest of the Pintool, as well as the instrumented program, will halt until the current Pintool operation is finished. Disk I/O is one example of such an operation. All other execution will halt until the I/O is finished. There is a wrinkle in how this works that affects our Pintool. Pin executes analysis functions in parallel [23]. As our Reservoir Sampling tool reads from an instrumented function's Reservoir, it is still possible that Pin calls that function's corresponding analysis object and overwrites some of the data currently being read. The larger the sampling percentage, the more data that has a chance of being overwritten. There would be no way to tell how representative the results would be due to data being overwritten.

To solve this problem, we use busy waiting to prevent a Reservoir from being simultaneously written to and read from. When the Pintool begins to read from a function's Reservoir, that function's analysis object is prevented from writing to the Reservoir until reading is finished.

The final challenge was attempting to parallelize our tool by adding a Reader component. When a given instrumented function's data is being written to disk, that function's analysis object is blocked to prevent overwriting data that has yet to be read. This of course causes a delay in the Pintool, as well as the instrumented application. Our idea was to create a Reader that ran on its own thread, and could handle disk I/O while the rest of the Pintool continued running. Because we wanted to keep overhead to a minimum, we did not want to use locking mechanisms. Instead, a double buffer was used. One buffer was used by the Pintool for writing results into memory, while the other buffer was read by the Reader and written to disk. This allows for simultaneous reading and writing without using any sort of locking, and without the risk of unread data being overwritten. The Reader is described in more detail later in Chapter 4.3.

4.2 Reservoir Sampling Implementation

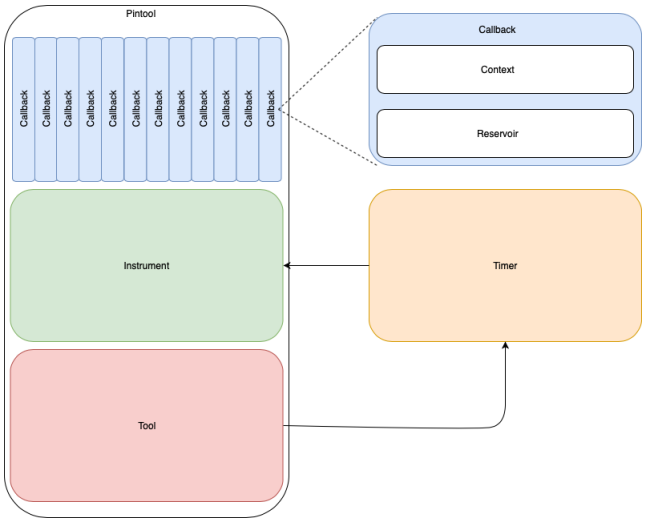

Figure 4.1.: Structure of the reservoir sampling tool. Each routine will have its own Callback. The Timer keeps track of time intervals.

Figure 4.1 shows the makeup of our reservoir sampling tool. Our tool contains a single Instrument. When a Pintool starts up, Pin analyzes the binary to find locations that need to be instrumented. Our tool's Instrument looks for all the function calls in the program's binary. The Instrument additionally creates a list of all of these locations. Each function has its own analysis code. The code for each analysis function is identical, but each function will have its own instance of the analysis object. In the binary, a Callback is injected before each function that will call that function's specific analysis code. Each function's Callback contains two main parts: the reservoir, and a Context. The reservoir is where sampled data is stored. In our tool, the reservoir's underlying data structure is a vector. The reservoir and the vector do not need to be the same size, as long as the vector is large enough to hold the entire reservoir. The Context is where the actual reservoir sampling code resides. The Context tracks and calculates the size of the reservoir, how many events have occurred, how many events have been sampled, and the skip constant.

Algorithm 1 shows the steps our tool takes when it program execution reaches an instrumented portion of the code.The Callback asks the Context if the next event should be sampled. If not, the Context decrements the skip constant and the Callback does nothing else. If the next event should be sampled, several steps are taken. If the reservoir's vector is at capacity, the vector's size is increased. The Callback asks the Context in which element of the reservoir the new sample should be stored. In the first stage of the reservoir sampling algorithm, this is simply the next empty element. Otherwise, the Context picks an element from the reservoir at random. The Context then recalculates the skip constant. In the regular version of the tool, there will be a possible busy wait between steps 2 and 3 of Algorithm 1. If the tool is currently reading from a given Reservoir, the corresponding analysis function will be made to wait until the reading is finished. In the multi-core version of the tool, there is no busy wait. There are separate read and write buffers, so there is no need to make the analysis function wait.

Algorithm 1 Algorithm for taking a sample

```
1:  procedure HANDLE_ANALYZE(address)
2:      if context → sample current then
3:          if vector is full then
4:              resize vector
5:          end if
6:          index = context → get next index
7:          reservoir[index] = address
8:          context → calculate next skip constant
9:      else
10:         context → decrement skip constant
11:     end if
12: end procedure
```

The Timer is an additional, non-Pin part of the program. Algorithm 2 shows the algorithm for how the Timer works. The Timer is what keeps track of time intervals. It runs on its own thread using Pin++'s threading API. Pin++ uses Pin's thread API to safely create Pintool threads. This is necessary, because it is not possible to use the Linux `pthread` library to create threads in a Pintool [23]. The Tool creates and starts the Timer during initial startup, and stops the Timer after the instrumented program has ended. After starting, the Timer thread goes to sleep for an interval of time. When it wakes up, it checks to see what woke it up. If it woke because the time interval expired, the Timer send a message to the Instrument and then go back to sleep. The Instrument then calls the `reset` function of each routines' Callback. The Callbacks write the contents of the reservoir to an external file, call their corresponding Context to reinitialize the state of the reservoir sampling algorithm, and fill the reservoir's underlying vector with zeros. When the instrumented program is finished, the Tool will stop the Timer by prematurely notifying it. The Timer will wake, see that the cause was not the time interval expiring, and perform one last notification.

4.3 Reader Implementation

We created a parallelized version of the tool. In the original tool, the code that wrote results to disk was part of the analysis code. During disk I/O, there was a risk that a given function's analysis object could be called while that function's Reservoir was being read, causing unread data to be overwritten. A busy wait was used to prevent the analysis object from writing to a Reservoir that was currently being read, but this caused runtime delays. In this version, a Reader was added to handle writing results to disk. Algorithm 3 shows how the Reader works. The Reader runs on its own thread. Each Callback now has a double buffer. One is for reading, the other for writing. During the initial Pintool setup, these buffers are registered with the Reader. In this version, when the Timer goes off, the Instrument now notifies each Callback to swap the read and write buffers, as well as updating the reservoir size of

Algorithm 2 Algorithm for Timer notification

```
    Timer awakens
2:  if Time interval expired then
        Timer → notify Instrument
4:      Timer sleep
    else
6:      Timer → notify Instrument
        Timer stops
8:  end if
    for all Callbacks do
10:     Instrument → notify Callback
        Callback → write contents to disk
12:     Callback → reinitialize algorithm state
        Callback → reinitialize reservoir with zeroes
14: end for
```

the write buffer. The Instrument then notifies the Reader that it can begin reading. By sequentially notifying the Callbacks to swap buffers and then notifying the Reader, we can make sure the Reader does not start reading until the buffers are ready. The Reader goes through its list of readers, checks the size of the reservoir to determine how many elements need to be handled, and writes the results to disk. The use of two buffers prevents unread data from being overwritten without the use of busy waiting or locking mechanisms.

To try and ensure the Reader could run uninterrupted, we also implemented a Scheduler that can request threads to be run on a specific CPU core. The Linux command `sched_setaffinity` is used to set the CPU cores a thread is allowed to run on. However, the kernel may impose restrictions on which cores a process can run on. There is no guarantee that a thread will be scheduled to a requested CPU core. This is a best-effort attempt to schedule the Reader thread on its own core [24]. The Scheduler is also used to schedule other Pintool threads. The main Pintool thread and the Reader thread are both scheduled on their own cores. Any other threads created by the instrumented program are scheduled on a set of cores round-robin style. The cores used for scheduling are designated in the Tool portion of the Pintool.

Algorithm 3 Algorithm for threaded Reader

```
    Timer → notifies Instrument
    for all Callbacks do
3:      Callback → update buffer reservoir size
        Callback → swap read and write buffers
        Callback → reinitialize buffer with zeroes
6:      Callback → reinitialize algorithm state
    end for
    Instrument → notify Reader to start
9:  for all Buffers do
        Reader → get reservoir size
        Reader → write buffer contents to disk
12: end for
```

5 EXPERIMENTAL RESULTS

This chapter discusses the setup of our experiments with our reservoir sampling tool. It describes the environment the experiments were run in, as well as the details of the experiments themselves. Finally, the results of the experiments are discussed.

5.1 Experiment Setup

Pin++ uses features of C++11. The most recent version of Pin, Pin 3 uses an OS-agnostic component called the PinCRT, which provides APIs that are used in place of aspects of native OS system calls. The CRT also uses its own C-runtime layer and C++ runtime. These APIs allow Pintools to avoid dependency on a specific OS or set of tools. However, at this time, the CRT is incompatible with C++11 [25].

Because of this incompatibility, this implementation uses Pin 2.14, the most recent version of Pin 2. Pin 2 does not use the PinCRT, and is thus compatible with C++11. The test system used Ubuntu as the OS. The most recent versions of Ubuntu use the Linux 4 kernel. However, when attempting to use Pin 2 with the Linux 4 kernel, Pin will display an error message stating that Pin 2 is incompatible with Linux 4. Instead, we needed to use Linux 3. The latest LTS version of Ubuntu that uses the Linux 3 kernel is Ubuntu 14 [26]. The testing was performed on Ubuntu 14 with the Linux 3.13 kernel installed.

The experiments were run on the System Integration Lab at IUPUI, which is an instance of Emulab [27]. Emulab creates nodes using operating system images. These nodes act as individual systems, and multiple nodes can be configured to work together and emulate many types of networks. An image was made from the Ubuntu installation, and was used for developing and testing our tool. Two types of nodes were available in our test system. The first type of node ran on an AMD Opteron

4130 processor with four cores and a maximum clock speed of 2.6 GHz, and had 8GB of RAM. The second type of node contained four AMD Opteron 6272 processors with a max clock speed of 2.6 GHz. Each of these CPUs had eight cores, and each of the cores could support two threads. Overall, this system had 64 cores available. It had 32GB of RAM.

To test our tool, we wanted to mimic conditions of actual program use. Performance was tested using Sysbench [28], a multi-threaded benchmarking framework. Among its features is the ability to benchmark MySQL databases. Specifically, Sysbench can run an online transactional process (OLTP) benchmark on a MySQL database. First, a MySQL table is created. In our testing, the table size used was 1,000,000 entries. The benchmark can be run for either a maximum amount of time, or a maximum number of requests [29]. We chose to run the benchmark multiple times with increasingly larger numbers of maximum requests. The benchmark was instrumented with our tool, and runtime to completion was measured.

The experiments were run on one of the four-core nodes, and on the sixty-four-core node. Both the regular version of our tool, and the multi-core version with the Reader were tested. The tests were run using a Bash script. The Linux `date` command [30] was used to track runtime. The current Unix epoch time was recorded before the tool started and after it finished, and the difference in time was recorded as the runtime in milliseconds.

Our first experiment measured performance of the single-core version of our tool on both the four-core and sixty-four-core machines. Several other Pintools were also tested to compare their performance against our tool. The benchmark was run in the following ways:

1. **Baseline.** The benchmark was run with no instrumentation. This allows us to determine how much overhead the instrumentation methods add.

2. **Exhaustive sampling.** Every single event that can be recorded is recorded. The other methods can be compared to this to see how much less overhead they cause.

3. **Percentage-based sampling.** For every event, a percentage is calculated to see if the event is sampled.

4. **Constant sampling.** Every *nth* event is sampled.

5. **Reservoir sampling.** This is our tool. It aims to reduce instrumentation overhead while also providing system visibility.

For the percentage-based, constant, and reservoir sampling methods, a sampling percentage of 20% was arbitrarily picked for testing. Exhaustive sampling samples 100% of events, and the baseline is not instrumented.

Our second experiment measured performance of the multi-core version of our tool which uses a Reader to attempt to parallelize writing results to disk. Again, this tool is tested on both the four-core and sixty-four-core machines.

A third experiment looked at how increasing the percentage of samples affected added runtime.

5.2 Reservoir Sampling Pintool Experiment

Figure 5.1 shows the results of the benchmark on the 4-core machine. The figure shows time in seconds, and the number of MySQL requests the benchmark performed. As expected, the baseline test with no instrumentation performed the fastest, and the exhaustive sampling test which sampled every event was the slowest. The other three methods, constant, percentage, and reservoir sampling, are all somewhere in-between. All three of these methods performed similarly.

Figure 5.2 shows just these three methods. Constant sampling performs somewhat better than the other two methods. Reservoir sampling is slightly slower than

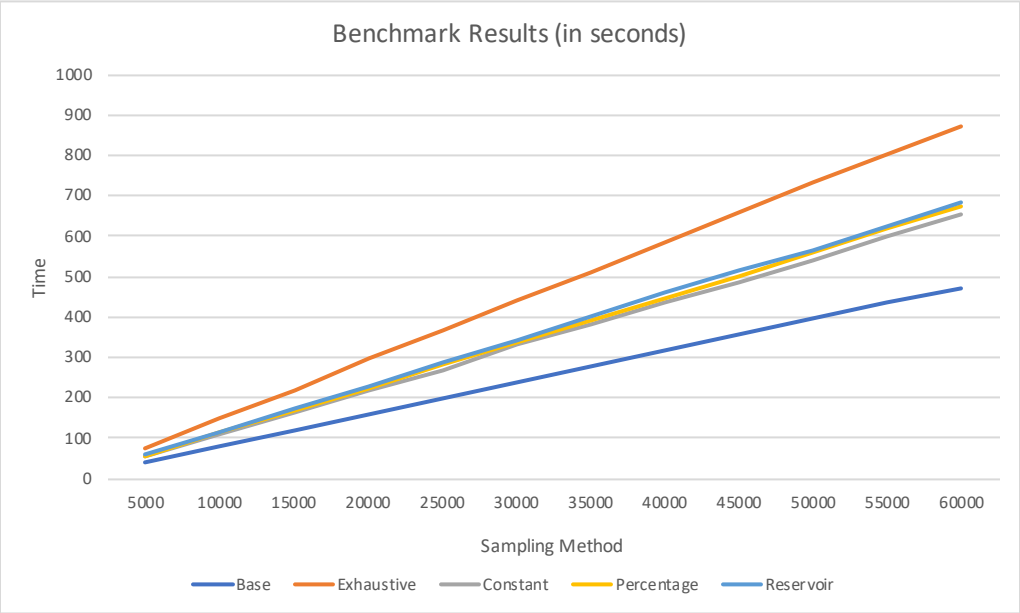

Figure 5.1.: Results of all sampling methods on 4-core machine

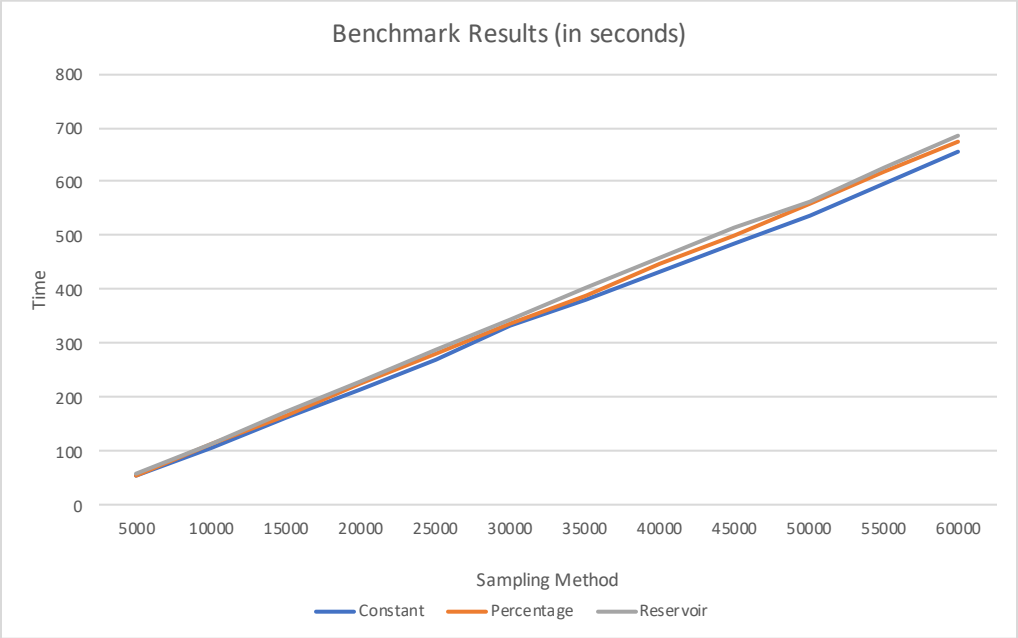

Figure 5.2.: Results of constant, percentage, and reservoir sampling on 4-core machine

percentage-based sampling. While Reservoir sampling performs sightly slower, it does give us the desired system visibility the other methods do not.

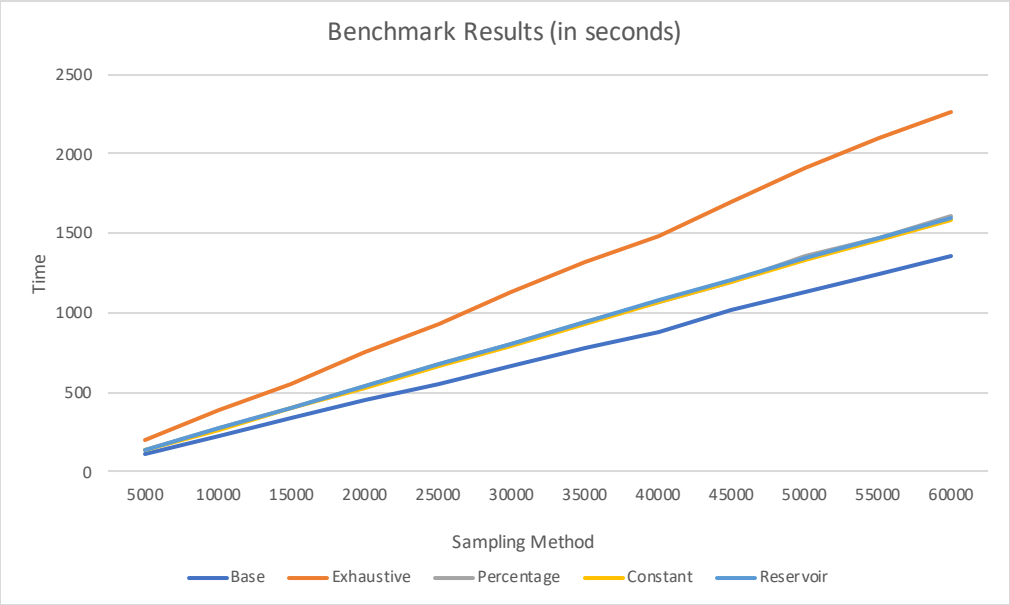

Figure 5.3.: Results of all sampling methods on 64-core machine

Figure 5.3 shows the results of testing on a 64-core processor. For unknown reasons, the tests took roughly two to three times longer on the 64-core machine. However, the difference in time between the three sampling methods and exhaustive sampling is even greater than it was on the 4-core processor. Again, constant, percentage-based, and Reservoir sampling all performed similarly. Constant sampling was slightly faster, and reservoir sampling was similar to, or slightly faster, than percentage-based sampling.

The results show that all three sampling methods are considerably faster than exhaustive sampling, with the difference being even greater on the 64-core processor. On the 4-core processor reservoir sampling was somewhat slower than constant sampling, which was expected. However, it was also slightly slower than percentage-based sampling, even though the point of reservoir sampling is to be computationally efficient. However, it is important to remember that the reservoir algorithm is being

performed for every single routine in the program. The Pintool counted 3,334 different routines that each had their own Callback with Reservoir and Context. The reservoir sampling calculations were being performed for all 3,334 routines. Additionally, this method of sampling ensures visibility of every routine. If visibility is important, the slight additional overhead may be worth it. On both processors, the time difference between exhaustive sampling and reservoir sampling increased the longer the benchmark ran. For long-running processes, this is ideal. The longer the process runs, the greater the amount of time between exhaustive sampling and reservoir sampling.

5.3 Multi-core Reservoir Sampling Pintool Experiment

This experiment tests the performance of our tool with the addition of the multi-core Reader component. The Reader reads sample data from a double buffer while the tool continues to write new data, so ideally we should see some kind of performance increase. The tool was tested on both the four-core and sixty-four-core machines. The main thread of the tool and the Reader are assigned to their own cores. Application threads spawned by the benchmark will be assigned to other cores in a round robin fashion.

Figure 5.4 shows the results of the multi-core tool on the 4-core system. The multi-core version not only performs better than the basic reservoir sampling version, but it also performs better than the percentage-based version, and even the constant version. As mentioned before, Pintools are generally single-threaded. The percentage-based and constant Pintools will pause execution while results are being written to disk. The multi-core version does not have this issue, and continues sampling during disk I/O, allowing it to perform better. Figure 5.5 shows only the base, reservoir, and multi-core methods to better compare reservoir and multi-core performance.

Figure 5.6 shows that on the 64-core system, the multi-core performance is even more improved. The multi-core version clearly performed better than the other three versions.

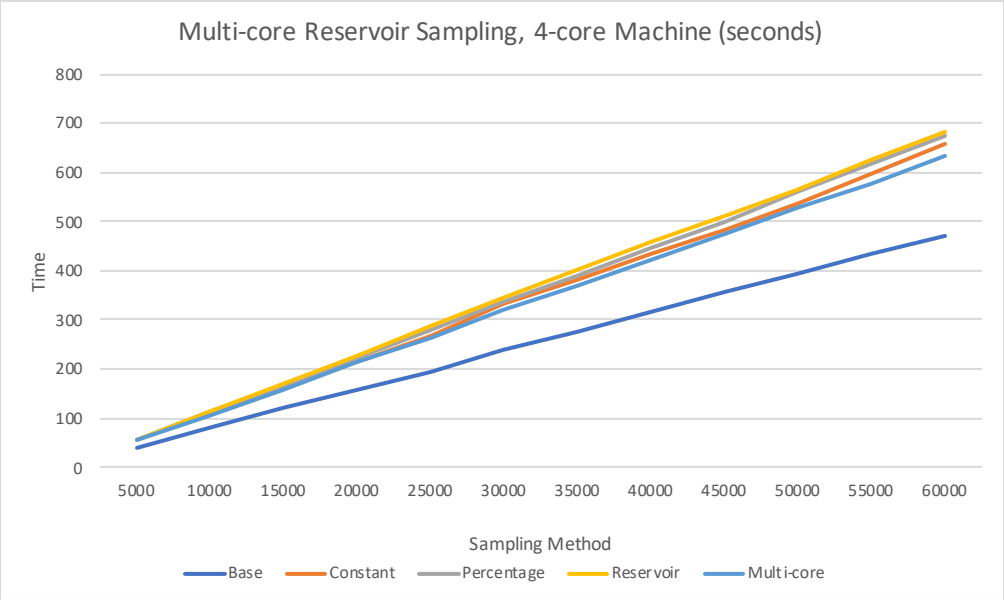

Figure 5.4.: Results of Multi-core Reservoir Sampling on 4-core machine

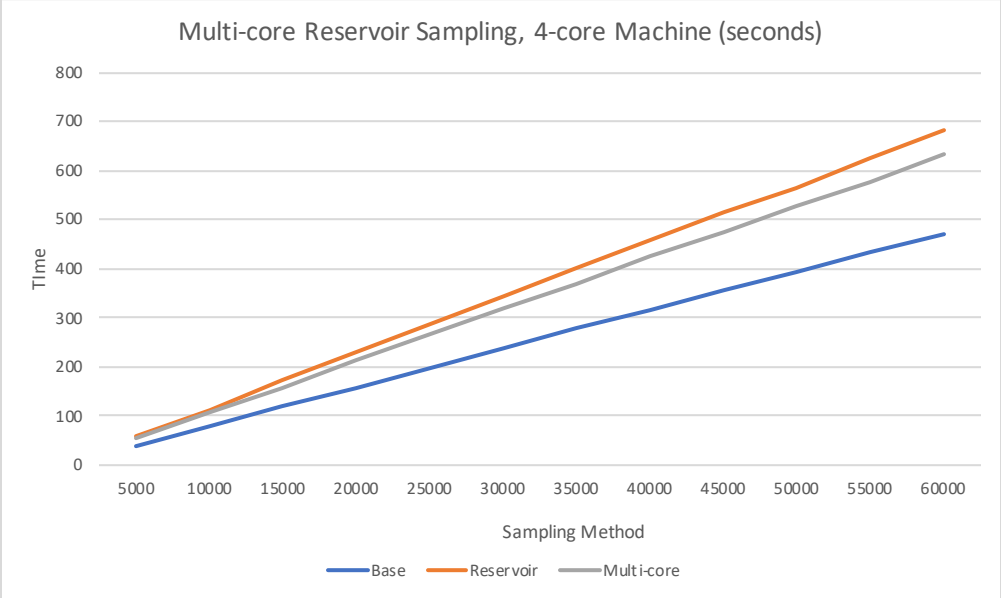

Figure 5.5.: Results of Multi-core Reservoir Sampling on 4-core machine

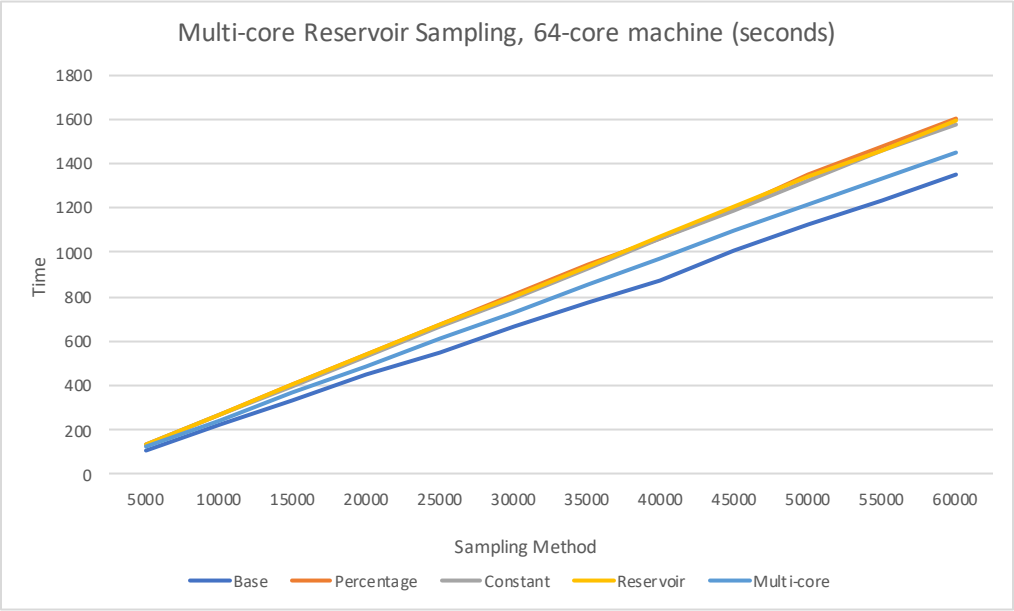

Figure 5.6.: Results of Reservoir Sampling methods on 64-core machine

In addition to considering absolute time taken, we can also look at how much time instrumentation methods take compared to the baseline as a percentage. Ideally, this percentage would not increase as runtime increases. Our approach is meant to instrument long-running processes. If the percentage of extra overhead were to increase as runtime increases, the tool would be unsatisfactory for our purposes.

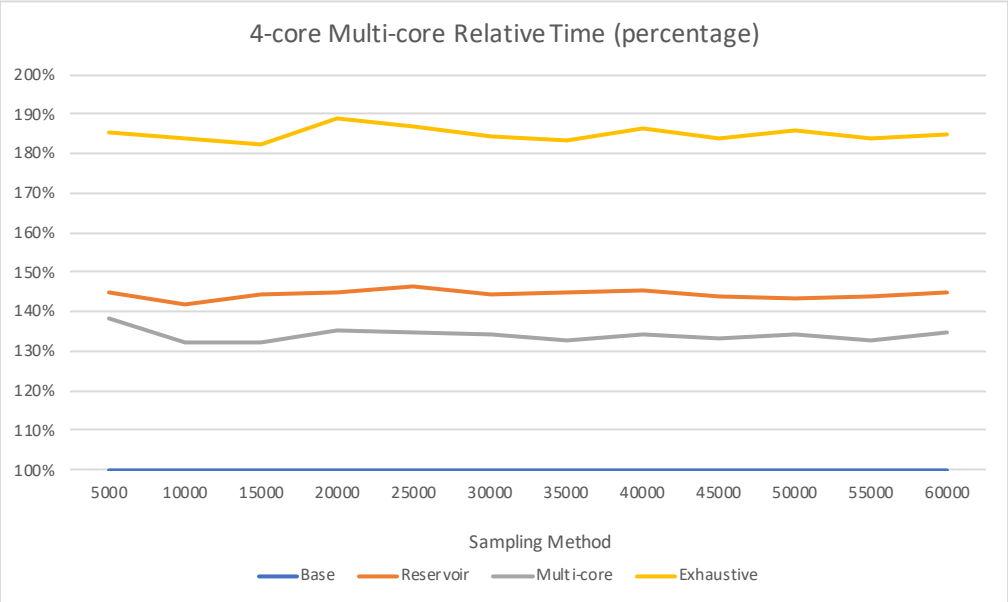

Figure 5.7.: Runtimes relative to the base case on 4-core machine

Figures 5.7 and 5.8 show instrumentation times as a percentage compared to the baseline of running the benchmark on its own. In 5.7 we see that exhaustive sampling adds an average of 85 percent extra runtime. Reservoir sampling reduces that extra runtime to an average of 44 percent. That is nearly 50 percent less overhead. The multi-core version further reduces added overhead to 34 percent, a reduction of 60 percent.

In 5.8 the difference is even more stark. After an initial spike, exhaustive sampling runtime settles into 70 percent extra overhead. Reservoir sampling increases runtime by an average of 20.5 percent. The multi-core version averages just under 10 percent extra overhead, which is a reduction of more than 85 percent compared to exhaustive sampling. Additionally, on both the four-core and the sixty-four-core machine, the

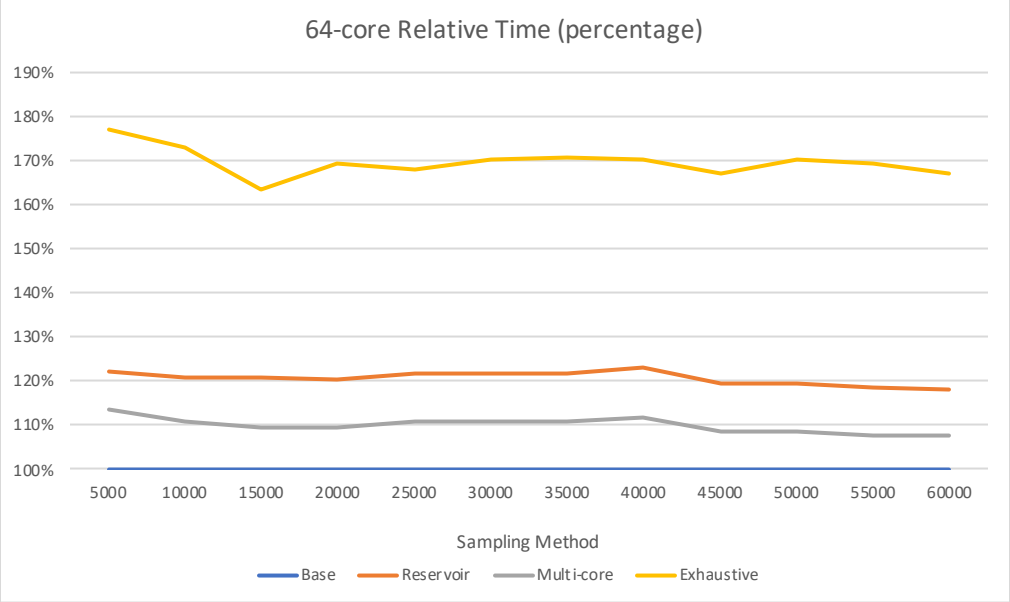

Figure 5.8.: Runtimes relative to the base case on 64-core machine

amount of extra overhead incurred by reservoir sampling stays roughly even. This suggests that our approach scales with both the amount of the the instrumented program runs, and with the number of cores in the system.

Part of the motivation behind this experiment was to create a tool that could send the resulting samples to a remote system. One core would be dedicated to the Reader that would handle sending the data remotely, leaving the other cores to focus on running the Pintool and application. One benefit of doing this is that the remote system could handle processing and storage to offload some of the additional overhead. To take this a step further, the remote machine could also analyze the instrumentation data, and provide feedback to allow the host to adjust its performance. The current Reader component in our multi-core tool is the first step towards sending data to a remote system. Instead of writing data to a local disk, the Reader would instead send it over a network. We wondered just how much time the Reader was spending on I/O.

We ran another experiment on the sixty-four-core machine. The multi-core reservoir reader tool was run to measure how much time the Reader spent reading, and how much time it spent idle. For each time interval, a timestamp was recorded after the Reader had been notified by the Instrument. The time between these timestamps determined total time for the Reader for an interval. These times were accumulated as the program ran. A timestamp was recorded directly before and after the Reader started and finished reading, and determined how much time the Reader spent reading for that interval. Read times were also accumulated. After the benchmark finished, the total and reading times were reported.

Figure 5.9 shows the results. At 20 percent sampling, the Reader actually spends most of its time idle. Figure 5.10 shows what percent of time the Reader spends idle. The Reader spends eighty-five percent of its time waiting for the interval to end so it can start its next round of disk I/O.

This large amount of idle time suggests that if the Reader were changed to send instrumentation data to a remote server, the Reader would have adequate time. Even

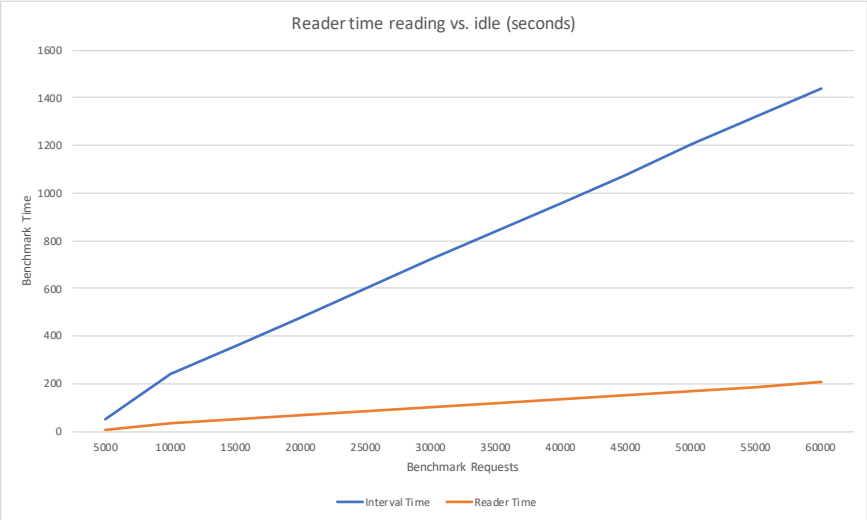

Figure 5.9.: Reader timing on 64-core machine

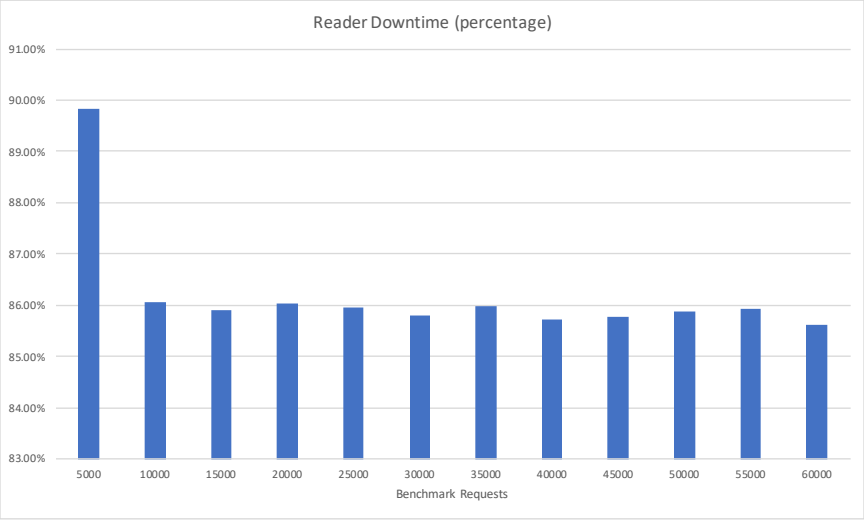

Figure 5.10.: Reader downtime on 64-core machine

without considering a remote server, the Reader could possibly do more work on the local machine. The downtime could possibly be used to do some of the analysis suggested for the remote machine, or the sampling percentage could possibly be raised without affecting runtime.

5.4 Sampling Percentages Experiment

The third experiment looks at how increasing sampling percentages will affect the runtime of our Pintools. With the constant, reservoir, and percentage-based Pintools, the expectation is that higher sampling percentages will cause higher runtimes. Considering the amount of time we found the Reader to be idle, it may be possible to increase the sampling rate of the multi-core version without greatly affecting runtime.

To measure this, we considered the performance of the various methods compared to the base method, in the same manner as we did in Figures 5.7 and 5.8. These figures show the extra overhead of the reservoir and multi-core Pintools, sampling at 20 percent, compared to the base case. From this information, we can find the average amount of extra runtime added by these methods at 20 percent sampling. For this experiment, the constant, percentage-based, reservoir, and multi-core methods were tested. The tools were run at 20, 40, 60, 80, and 100 percent sampling. The average amount of extra overhead was found for each method at each level of sampling. Table 5.1 shows the results.

For reference, the average overhead added by exhaustive sampling, which is 100 percent sampling, was 85 percent. With higher sampling levels, constant, percentage-based, and reservoir sampling all took longer as expected. At 100 percent sampling, these three methods were all slower than exhaustive sampling. Interestingly, the runtime of multi-core sampling stayed fairly stable regardless of the sampling level. Timestamps were used to ensure that during each interval the Reader did, in fact, finish reading before the end of the interval.

Table 5.1.: Average additional overhead on 4-core system

	Sampling Percentage				
Method	20%	40%	60%	80%	100%
Constant	37%	55%	84%	84%	86%
Percentage	42%	54%	67%	79%	90%
Reservoir	44%	57%	67%	78%	86%
Multi-core	34%	35%	35%	33%	28%

Table 5.2.: Average additional overhead on 64-core system

	Sampling Percentage				
Method	20%	40%	60%	80%	100%
Constant	19%	37%	71%	71%	71%
Percentage	21%	33%	44%	57%	70%
Reservoir	21%	34%	46%	59%	69%
Multi-core	10%	10%	12%	10%	9%

Table 5.2 shows the results of this experiment on the 64-core system. The average overhead added by exhaustive sampling was 70 percent. Again, constant, percentage-based, and reservoir sampling methods have increased overhead with higher sampling percentages, and multi-core sampling overhead stays relatively stable as sampling percentage increases. These results show that it is possible to use the multi-core Pintool to increase the sampling percentage without greatly affecting overall runtime.

5.5 Memory Usage

One final aspect of approach we examined was how much memory our Pintools used, and how that amount of memory changed over time. The Reservoirs in our tools were implemented using C++ vectors. During Pintool execution, when the Reservoir size exceeds its vector's capacity, the vector is doubled in size. When smaller increases were tried, i.e. increasing vector capacity by 1.5 times, the additional resizing operations caused crashes due to memory allocation failures. We were curious to see how this affected memory use during the execution of the Pintool.

We measured memory use by reading the contents of the `/proc/self/statm/` file. This file contains information about a program's memory usage [31]. This file is updated as a program runs. From this file we periodically read the total program size. After each time interval, the current memory use was recorded into a log file. We tested both the regular and multi-core versions of the Reservoir sampling tool, on both the 4-core and 64-core systems.

We found that the memory usage is mostly stable during the runtime of the Pintool. The difference in memory use when comparing the 4-core and 64-core systems was minimal, and the total runtime made very little difference. The sampling percentage did matter somewhat. The results from the 4-core system are shown in table 5.3.

The multi-core version of the tool uses a bit more than twice the memory the regular version, due to using two buffers per reservoir. The Pintools tended to allocate

Table 5.3.: Memory use of Pintools at various sampling levels

	Sampling Percentage				
Pintool	20%	40%	60%	80%	100%
Regular	473 MB	477 MB	480 MB	484 MB	488 MB
Multi-core	969 MB	977 MB	988 MB	996 MB	1008 MB

extra memory when the program first began execution, reaching max memory by the second or third sampling interval. Increasing runtime did not affect memory use, which indicates the Pintools' memory use should be stable even when instrumenting long-running processes.

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

This thesis described a method of reducing software system instrumentation overhead. By creating an instrumentation tool that uses reservoir sampling, we were able to reduce overhead by collecting only a sample of the available data. Reservoir sampling ensures that even though we do not know in advance how many events will occur, every event has the same probability of ending up in the final results. Sampling each function separately ensured we achieved full system visibility, with every function being represented in the sample. A multi-core approach was used to parallelize disk I/O, allowing for simultaneous reading and writing of samples. On our four-core system, exhaustive sampling increased runtime by an average of 85%. Our method increased runtime by 34.1%. On our sixty-four-core system, exhaustive sampling increased runtime by an average of 69.9%. Our method's added overhead was only 9.9%.

From our experiments with our approach to real-time system instrumentation, we have learned the following:

- We can successfully use reservoir sampling in instrumentation to reduce added runtime overhead. We can do this in such a way that we achieve full system visibility, as well.

- On our four-core system, the added runtime from reservoir sampling was slightly higher than the runtime from constant sampling or percentage-based sampling. If full system visibility is desirable, this small amount of extra overhead could be worth it.

- On our sixty-four core system, reservoir sampling was slightly faster than percentage-based sampling, but slightly slower than constant sampling.
- Our results show that our approach scales well on both the four-core and sixty-four-core systems. The results also show our approach scales well as the runtime of the instrumented program increases.
- Our attempt at creating a multi-core instrumentation tool that parallelized disk I/O made a difference. The multi-core version of our reservoir sampling tool performed better than either the percentage-based or consatant versions, on both the four-core and sixty-four-core systems.
- We found that at 20 percent samlpling, the parallelized portion of the tool spent most of its time idle, suggesting it could be used for additional tasks.
- We were able to increase the percentage of sampling of the multi-core tool without greatly increasing runtime.

6.2 Future Work

There are additional changes that could possibly improve performance of Pintools using reservoir sampling. Another potential change is how the sampled data is handled. Our approach simply writes the results to an external file on the hard drive. Another option would be to stream the sampled data to a remote system. This could have two benefits. First, it offloads work from the system running the instrumented program, allowing the program to run as quickly as possible. If a program has a lot of disk I/O, the Pintool might cause resource contention when it writes data to disk. Sending the data to a different system would prevent that. Secondly, the other system could possibly analyze the results, and send feedback to the original system. The feedback could be used to tune the performance of the original system. We will be conducting additional work on this idea in the future.

We used sampling to reduce the amount of data being collecting, thus reducing the amount of overhead added by instrumentation. Another possible way of reducing the amount of data collected is to target specific parts of the program for instrumentation. Our approach instruments and samples every routine in the underlying program. If only specific parts of the program are of interest, the unwanted parts can be filtered out. Pin is capable of performing such filtering. Only the desired parts will be instrumented and sampled, reducing runtime and providing more relevant results.

There are two variables that could be adjusted by the user: interval length, and the threshold value used in the reservoir sampling algorithm. Our approach used default values for these variables, but changing these values is likely to affect performance. It might even be possible to improve performance with the correct settings. However, there are many other issues that affect performance such as the characteristics of the system, or the instrumented program itself. For example, we saw from our results that running the same tests on a 4-core processor gave different results than running them on a 64-core processor. If it can be determined that adjusting interval length and the threshold value can improve performance, it would be tedious to figure out the correct combination of values for a given system. Instead, it would be better to have a way for the Pintool to be self-adjusting. Trial runs would measure performance with different settings, and attempt to find the ideal settings for a given setup.

REFERENCES

REFERENCES

[1] J. C. Huang. Program instrumentation and software testing. *Computer*, 11(4):25–32, April 1978.

[2] Shyh-Kwei Chen, W. K. Fuchs, and Jen-Yao Chung. Reversible debugging using program instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, Aug 2001.

[3] D. Mahrenholz, O. Spinczyk, and W. Schroder-Preikschat. Program instrumentation for debugging and monitoring with aspectc++. In *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pages 249–256, April 2002.

[4] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead interactive debugging via dynamic instrumentation with dise. In *11th International Symposium on High-Performance Computer Architecture*, pages 303–314, Feb 2005.

[5] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 198–208, Washington, DC, USA, 2007. IEEE Computer Society.

[6] S. Elbaum and M. Diep. Profiling deployed software: assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, April 2005.

[7] Edu Metz and Raimondas Lencevicius. Efficient instrumentation for performance profiling. *CoRR*, cs.PF/0307058, 2003.

[8] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, SE-5(3):226–236, May 1979.

[9] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[10] Kim Hazelwood. Dynamic binary modification: Tools, techniques, and applications. *Synthesis Lectures on Computer Architecture*, 6(2):1–81, 2011.

[11] Pin, a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[12] Dynamorio. `http://dynamorio.org`.

[13] Valgrind. `http://valgrind.org`.

[14] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *In WBIA Workshop at ASPLOS*, 2006.

[15] James H. Hill and Dennis C. Feiock. Pin++: An object-oriented framework for writing pintools. *SIGPLAN Not.*, 50(3):133–141, September 2014.

[16] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[17] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[18] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *N 4TH ACM WORKSHOP ON FEEDBACK-DIRECTED AND DYNAMIC OPTIMIZATION*. ACM, December 2001.

[19] Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. Int. Symp. Code Generation and Optimization CGO '07*, pages 209–220, 2007.

[20] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. Pipa: Pipelined profiling and analysis on multicore systems. *ACM Trans. Archit. Code Optim.*, 7(3):13:1–13:29, December 2010.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Notes*, 40:190–200, June 2005.

[22] Intel. Pintools instrumentation granularity, pin 2.14 manual. `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html#GRAN`.

[23] Intel. Instrumenting multi-threaded applications, pin 2.14 manual. `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html#MT`.

[24] *sched_setaffinity(2) Linux Programmer's Manual*, 09 2013.

[25] Intel. Pintools information and restrictions, pin 3.5 manual. `https://software.intel.com/sites/landingpage/pintool/docs/97503/Pin/html/index.html#RESTRICTIONS`.

[26] Ubuntu kernel release schedule. `https://wiki.ubuntu.com/Kernel/Support#Ubuntu_Kernel_Release_Schedule`.

[27] The University of Utah. Emulab - Network Emulation Testbed Home. `http://www.emulab.net/`.

[28] Sysbench, scriptable database and system performance benchmark. `https://github.com/akopytov/sysbench`.

[29] *sysbench(1) sysbench User Manual*, 02 2014.

[30] Free Software Foundation. *date(1) Linux User Commands*, 01 2015.

[31] *proc(5) Linux Programmer's Manual*, 03 2019.