Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Mike Du

Entitled

Realtime Dynamic Binary Instrumentation

For the degree of Master of Science

Is approved by the final examining committee:

James H. Hill
Chair

Rajeev R. Raje

Mihran Tuceryan

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): James H. Hill

Approved by: Shiaofen Fang 4/1/2016

Head of the Departmental Graduate Program Date

REALTIME DYNAMIC BINARY INSTRUMENTATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Mike Du

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2016

Purdue University

Indianapolis, Indiana

To my family.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Dr. James H. Hill, for his guidance, advice, and knowledge, without whom, this work would not have been possible. I would also like to thank Dr. Rajeev R. Raje and Dr. Mihran Tuceryan for being on my thesis defense committee. Another thanks goes out to Dennis Feiock and Manjula Peiris for their assistance in answering various questions and helping debug some issues I faced. Finally, I want to thank my family and friends for their support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

ABSTRACT

Du, Mike. MS, Purdue University, August 2016. Realtime Dynamic Binary Instrumentation. Major Professor: James H. Hill.

This thesis presents a novel technique and framework for decreasing instrumentation overhead in software systems that utilize dynamic binary instrumentation. First, we introduce a lightweight networking framework combined with an easily extensible BSON implementation as a heavy analysis routine replacement. Secondly, we bind instrumentation and analysis threads to non-overlapping cpu cores—allowing analysis threads to execute faster. Lastly, we utilize a lock-free buffering system to bridge the gap between instrumentation and analysis threads, and minimize the overhead to the instrumentation threads. Using this combination, we managed to write a dynamic binary instrumentation tool (DBI) in Pin using Pin++ that is almost 1100 % faster than its counterpart DBI tool with no buffering, and less than 500% slower than a similar tool with no analysis routine.

1 INTRODUCTION

Dynamic binary instrumentation (DBI) [1] is a powerful tool that enables users to obtain detailed runtime information about a program. Uses for these tools include debugging [2,3], cache simulation [4], parallel program analysis [5] and vulnerability detection [6], among others. Examples of DBI frameworks include, but are not limited to Pin [7], Valgrind [8], and DynamoRIO [9].

A benefit of DBI is that it can be run on any application without the need for the source code being available. This enables users to analyze and modify any program without being dependent on the source code. It also allows users to collect more specific data about their programs compared to using special hardware. However, this comes at a cost. Usage of DBI typically incurs a large performance impact [3,10–12]. As research and experience has shown, a program undergoing DBI can run anywhere from a few percent slower, to hundreds, or even thousands of times slower. The magnitude of performance degradation, however, depends heavily on the level and type of instrumentation and analysis performed on the program under instrumentation. For example, instrumenting every instruction in a program will have greater impact on performance when compared to instrumenting every function call.

Many techniques have been proposed to minimize this overhead DBI has on its programs undergoing instrumentation. These techniques range from buffering instrumented data and analyzing on a separate thread [3,11] to forking the entire process and instrumenting the clone [12]. The common theme for each technique is to take advantage of under-utilized cores on a system. This, approach, however, may still create unintended overhead if the original process is multithreaded. Moreover, it can be taxing in terms of processor or memory usage. Unfortunately, no DBI can escape this fact; it can only hope to reduce the effects.

To address the aforementioned shortcomings by using multiple cores to support DBI, we introduce *Realtime Binary Instrumentation System* (ReBIS), a lightweight, portable networking and buffering framework to Pin++ [13], a C++ framework for authoring analytical tools for Pin. The goal of ReBIS is two-fold. First, we aim to introduce a novel method for reducing instrumentation overhead while also creating the proper abstractions to enable users to simultaneously perform instrumentation on multiple systems. Second, we aim to allow the instrumented programs to communicate with each other and/or with a central server. The former allows instrumented programs to coordinate with each other, and the latter allows each program to offload analytical operations to resources outside of the instrumentation environment.

Our work can be separated into two parts. The first part focuses on creating a simple interface that enables users to perform network communications in a Pintool on both Linux and Windows systems. Because Pin is designed to work on these systems, that is where we focused our efforts. Our framework, however, can easily be extended to support other systems as needed. The second part consists of a lock-free buffering system designed for fast writes to minimize overhead. This buffering system is intended to bridge the communication gap between our instrumentation and analysis threads. Binding of the instrumentation and analysis threads to separate CPUs was also used to enable our analysis threads to process data more in realtime.

For an application that spawned 240 threads and using a buffer size of 50, ReBIS with CPU binding was able to outperform no binding by a factor of 385%. To achieve similar speeds with no binding, a buffer size of 500 was required. Compared to a similar Pintool that instead of buffering the data and then sending, simply performed a send of each piece of data, ReBIS was 1098% faster.

1.1 Thesis Organization

This thesis is organized in the following manner. Chapter 2 will discuss work done by other researchers to help lessen instrumentation overhead. Chapter 3 provides some background of the tools and specifications we used. Chapter 4 discusses preliminary work, the design of the system, and various implementation details. In chapter 5, a description of our tests as well as their results will be displayed, with a discussion and lessons learned. Lastly, Chapter 6 provides concluding remarks and future research directions.

2 RELATED WORK

This chapter compares and contrasts our work to other similar techniques. In particular, we investigate techniques in DBI that use multiple processors and data buffering to reduce instrumentation overhead and improve performance.

2.1 Taking Advantage of Unused Processors

A recent trend in decreasing instrumentation overhead is to perform as much instrumentation and analysis as possible on separate threads with hopes that the application under instrumentation is not fully utilizing all of the CPUs. This assumption is generally safe to make, because few applications are optimized enough to use, or require the use of all of the cores on a processor [11]. Thus, many techniques involve forking the original process and instrumenting the forks so the original application can run unhindered.

For example, one such technique is called Shadow Profiling [12]. Shadow Profiling works by forking the application at various points of execution, and then performing instrumentation on those forked shadow processes. The profiling tool uses a sample size variable and a load variable to determine how many instructions each shadow process should execute and how many concurrent shadow processes should exist, respectively. This allows the user to control both coverage and overhead of the instrumentation. Using various combinations of sample size and load, Moseley et al. (2007) managed to obtain under 1% overhead with lower sample sizes and loads, and up to 19% overhead with higher values.

Another approach is called SuperPin [14]. This technique did not achieve as low overhead as Shadow Profiling, but guarantees complete coverage. Moreover, each slice (or shadow process) in SuperPin is spawned at points determined by the tool, rather than by the user. SuperPin can guarantee complete coverage by registering signals each time a new slice is spawned, and ensuring that the previous slice knows exactly when to terminate. Using this technique, SuperPin typically managed to outperform native Pintools by 200% to 600%, depending on the application being instrumented—typically with overhead less than 100%.

While these techniques managed to obtain very low application impact, they generally do not solve the problem of trying to instrument an application which requires most, if not all, of a system's processing resources. In light of this, further techniques were introduced to reduce instrumentation overhead for such applications.

2.2 Data Buffering

In the context of Pin DBI, prior research has focused on creating faster buffers for decreasing overhead. For example, Upton et al. [15]. introduced a new fast buffering application programming interface (API) for Pin. This buffering API achieved approximately a 4X speedup compared to the fastest buffering system previously available in Pin.

In an effort to reduce the instrumentation time of DBIs, many buffering techniques were implemented to enable a batch-analysis of collected data. In particular, new systems for separating the collection and analysis of data onto separate threads were created. Using such systems, extra overhead from forking the original process into separate processes is removed. Instead, it focuses on reducing overhead by buffering the collected data, and the subsequent batch-processing on a separate thread.

Cache-friendly Asymmetric Buffering (CAB) [11] introduces a new buffering system based upon a cyclic buffer. CAB utilizes two main principles to minimize overhead: (1) writing to the buffer should be as fast as possible; and (2) there should be no contention between producer and consumer threads. This differs slightly from our work in that we do allow some contention between instrumentation (producer) and analysis (consumer) threads to ensure complete coverage; whereas, CAB will over-write existing data if the consumers cannot catch up to the producers. To prevent this case from occurring, CAB also implements a sampling mode where consumers sample from the buffer rather than consume the entire buffer. Our framework mitigates this problem by implementing a lightweight, universally usable analysis routine allowing the overall runtime to be more heavily impacted by the instrumentation rather than analysis.

Pipelined Profiling and Analysis (PiPA) [3] is another system that performs analysis in parallel to the execution of the application. PiPA works by moving collected profiles to a processing thread, and the distributing the profiles to multiple analysis threads for analysis in parallel. This approach is in contrast to our work. This is because our approach does not utilize an intermediate processing thread, and typically requires a lower number of analysis threads.

Deferred analysis [16], built upon the buffering strategy, by introducing a novel adaptive analysis strategy, where based on the total CPU usage, analysis could occur on the same threads as the instrumentation, or be run on separate threads. This enables users to take advantage of underutilized CPUs when available, but also allowing users to avoid the extra overhead from inter-thread communication when underutilized CPUs are not available. While our work does not provide this adaptability, we were able to push most of the inter-thread communication overhead to our analysis threads using our CPU binding technique, which run on a separate CPU core than the data collection threads. In the end, our approach has less effect on overall runtime.

3 BACKGROUND

Before we discuss our technique for decreasing instrumentation overhead, we will first provide a brief overview of Pin and Pin++, the DBI frameworks that we used. This is followed by an overview of BSON, the data interchange format chosen to be used in our framework.

3.1 Pin

Pin [7] is a popular dynamic binary instrumentation framework developed by Intel. Tools created using Pin are called Pintools, which can be reused to instrument any program without needing to recompile a new tool. Pin can be run in two different modes, JIT (just-in-time), and probe mode. JIT mode runs the program in a virtual machine, and Pin inserts the instrumentation on an as-needed basis. In probe mode on the other hand, Pin inserts jump instructions to call the instrumentation functions where they are needed, so that the program can run natively. Probe mode, however, does not allow the insertion for very small units of a program, such as every instruction that JIT mode can. Not running in a virtual machine allows for a decrease in overhead, but results in a large reduction of the available API.

Listing 3.1 shows an example Pintool written in native Pin which counts the instructions of an application. Line 51 initializes the Pintool, while line 53 opens the file that will be written to when the Pintool terminates. Line 57 and the *Instruction* function tells Pin to insert the *docount* function before every instruction in the application. Line 61 tells Pin to insert the *Fini* function after the application terminates. Finally, line 64 allows the application being instrumented to start execution.

```
1  #include <iostream>
2  #include <fstream>
3  #include ”pin.H”
```

```cpp

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is
// executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction
// is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction,
    // no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
                  IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE,
    "pintool", "o", "inscount.out",
    "specify␣output␣file␣name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by
    // the application
    OutFile.setf(ios::showbase);
    OutFile << "Count␣" << icount << endl;
    OutFile.close();
}

INT32 Usage()
{
    cerr << "This␣tool␣counts␣the␣number␣of␣dynamic␣";
    cerr << "instructions␣executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary();
    cerr << endl;
    return -1;
}

```

```
int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument
    // instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application
    // exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Listing 3.1: An example Pintool in native Pin [17]

Pin++

Pin++ [13] is a C++ framework designed to improve upon native Pin by enhancing component reusability, simplifying Pintool creation, and in some cases, even reducing the instrumentation overhead. Pintools written using Pin++ can be run just like any other Pintool. Pin++ achieves this decreased overhead by utilizing template metaprogramming. Since Pin++ is written in C++, it allows for the creation of Pintools using a much higher level of abstraction and much more organized set of tools than compared to using native Pin.

Pintools written in Pin++ are split up into 3 parts, a `Tool`, a class which represents the tool itself, `Instruments`, which represent the various levels of granularity of instrumentation, and `Callbacks`, which represent the functions that can be inserted by the instruments. Such Pintools may have any number of `Instruments` and `Callbacks`, but must contain at least one `Tool` instance.

Listing 3.2 shows an example Pintool written in Pin++ that counts the instructions of an application. Its functionality is identical as the Pintool in Listing 3.1. Lines 47 to 71 define the `Tool`, which enables users to insert itself as a callback to certain points (line 52), such as during application termination, which calls the *handle_fini* function. The `Tool` is also typically where the `Instruments` are declared and instantiated; in this case, in the private member section on line 66. Lines 28 to 45 define an `Instrument`, in this case, an `Instruction_Instrument`, which represents an instruction-level granularity. Line 35 says that the `Callback` declared and instantiated on line 44, which is also a private member, should be inserted before every instruction. The `Callback` defined by lines 7 to 26 contains the *handle_analyze* function which the Instrument tells Pin to insert, as well as the state. Finally, line 78 acts as the main function, which initializes Pin, instantiates the `Tool`, and starts the application.

```
 1 #include "pin++/Callback.h"
 2 #include "pin++/Instruction_Instrument.h"
 3 #include "pin++/Pintool.h"
 4
 5 #include <fstream>
 6
 7 class docount : public OASIS::Pin::Callback
 8                        <docount (void)>
 9 {
10 public:
11   docount (void)
12     : count_ (0) { }
13
14   void handle_analyze (void)
15   {
16     ++ this->count_;
17   }
18
19   UINT64 count (void) const
20   {
21     return this->count_;
22   }
23
24 private:
  UINT64 count_;
};

class Instruction :
  public OASIS::Pin::Instruction_Instrument
         <Instruction>
{
public:
  void handle_instrument (const OASIS::Pin::Ins & ins)
  {
    this->callback_.insert (IPOINT_BEFORE, ins);
  }

  UINT64 count (void) const
  {
    return this->callback_.count ();
  }

private:
  docount callback_;
};

class inscount : public OASIS::Pin::Tool <inscount>
{
public:
  inscount (void)
  {
    this->enable_fini_callback ();
  }

  void handle_fini (INT32 code)
  {
    std::ofstream fout (outfile_.Value ().c_str ());
    fout.setf (ios::showbase);
    fout <<  "Count␣" << this->instruction_.count ();
    fout << std::endl;

    fout.close ();
  }

private:
  Instruction instruction_;

  /// @{ KNOBS
  static KNOB <string> outfile_;
  /// @}
};

KNOB <string> inscount::outfile_ (KNOB_MODE_WRITEONCE,
                                  "pintool", "o",
                                  "inscount.out",
                                  "specify_filename");

DECLARE_PINTOOL (inscount);
```

Listing 3.2: An example Pintool in Pin++ [18]

3.2 Binary JSON

Binary JSON (BSON) [19] is a data-interchange format based on JavaScript Object Notation (JSON) [20]. JSON provides user readability, combined with ease of marshalling and demarshalling. BSON improves upon JSON by providing faster scan speed with the addition of a length field to all variable length types (*i.e.*, documents, arrays, strings and binary types). The drawback of BSON is that in some cases, BSON encoding can use more space than JSON encoding. BSON improves upon JSON further by providing a broader set of explicit types. BSON even allows users to specify their own types instead of relying on user code to differentiate between different instances of the same type as long as the parser knows how to handle the new types.

A BSON-encoded document begins with a 4-byte integer, encoded in binary, which represents the size of the document, including the terminating null byte, 0x00. A document can then contain any number of elements, which consist of a byte representing the type of the element, followed by a null-terminated key, followed by the value of the element. For example, if one wished to encode foo:42, where foo is the key, and 42 is the value, one would obtain the following BSON-encoded document:

```
( 0x0E 0x00 0x00 0x00 ) 0x10 ( f o o 0x00 ) ( 0x2A 0x00 0x00 0x00 ) 0x00
```

The brackets are simply there for illustration purposes, and are not actually in the encoded document. The first set of brackets surround the document size, which in this case is 14, note the little-endian format. The next byte represents the element type, in this case an int32. The second set of brackets surround the null-terminated key, while the third surrounds the value, in this case 42. Lastly, the 0x00 byte terminates the document. In JSON, this would instead look like:

```
{foo:42}
```

In this case, the braces are part of the format.

Both Pin++ and BSON were integral parts of our framework, as the next Chapter discusses.

4 DESIGN AND IMPLEMENTATION OF REBIS

As seen in Figure 4.1, the proposed system consists of multiple parts, the buffering system, a data marshalling component, BiSON (our BSON implementation), and a networking portion. The goal of providing these pieces together is to provide a basic framework which users can use to author low-overhead Pintools with networking capabilities. In this chapter, we will discuss the design goals and methodologies of each of these components, followed by a closer look at some of the implementation details.

4.1 Preliminary Design Challenges

Initially, the main goal of this work was to design and implement a system that allowed the user could transmit data over a network using a Pintool. To this end, it was decided (1) to use a third party networking library to abstract away the low level socket details; (2) be usable in both Linux and Windows environments; and (3) and provide some higher level functionality, such as support for quality of service (QoS), multiple protocols, etc.

The first attempt was to use OpenDDS [21], which provides an expansive feature set, including portability, support for multiple transport protocols, and QoS, while also being easy to use. Unfortunately, attempting to compile a Pintool using OpenDDS showed that it was incompatible with Pin in a Windows environment. This was due to having to enclose all Windows-related code in a namespace, but doing so with OpenDDS resulted in issues that were too hard to resolve. We therefore sought a different solution.

The *Adaptive Communication Environment* (ACE) [22] was the next choice, which also happens to be the framework upon which OpenDDS is built. Like OpenDDS, ACE provides a rich set of features, and is easy to use. Unlike OpenDDS, ACE encloses all of its code inside its own namespace. Hoping it solved the OpenDDS problems, we therefore ported critical parts, such as its mutex and threading framework to Pin. This is because Pin provides its own application programming interface (API) for these concepts for proper instrumentation of the target program.

Fortunately, ACE was designed to be easily portable to other operating systems and/or runtime environments. Unfortunately, the port was not clean such that ACE mutexes and threads utilized the Pin API instead while other features used native system calls. Moreover, we learned that Pin was not able to load any Pintool that links to the ACE library. This is because ACE performed static initialization, which caused the Pintool to have a segmentation fault at load time. The problem was effectively a non-debuggable problem.

Given the previously failed attempts to use a third-party networking library, the choice was made to write a new, simple networking library. The only requirements for this library were that it be portable between Linux and Windows systems, and that it provide support for sending and receiving data over a TCP and/or UDP connection. The remainder of this chapter discusses the design and solution to this need.

4.2 Design of ReBIS

ReBIS operates by taking advantage of the multiple cores in modern processors. In particular, ReBIS designates one or more cores for use by the analysis thread, and pushes instrumentation concerns to the remaining cores. This approach is illustrated in Figure 4.1. As shown in Figure 4.1, each instrumentation thread also is associated with a single buffer. This allows for writing to the buffers as fast as possible because the data collection threads do not have to compete with each other trying to write to a buffer.

In an effort to make analysis threads as fast as possible, their task is only to take the instrumentation data and send it over the network using our custom networking

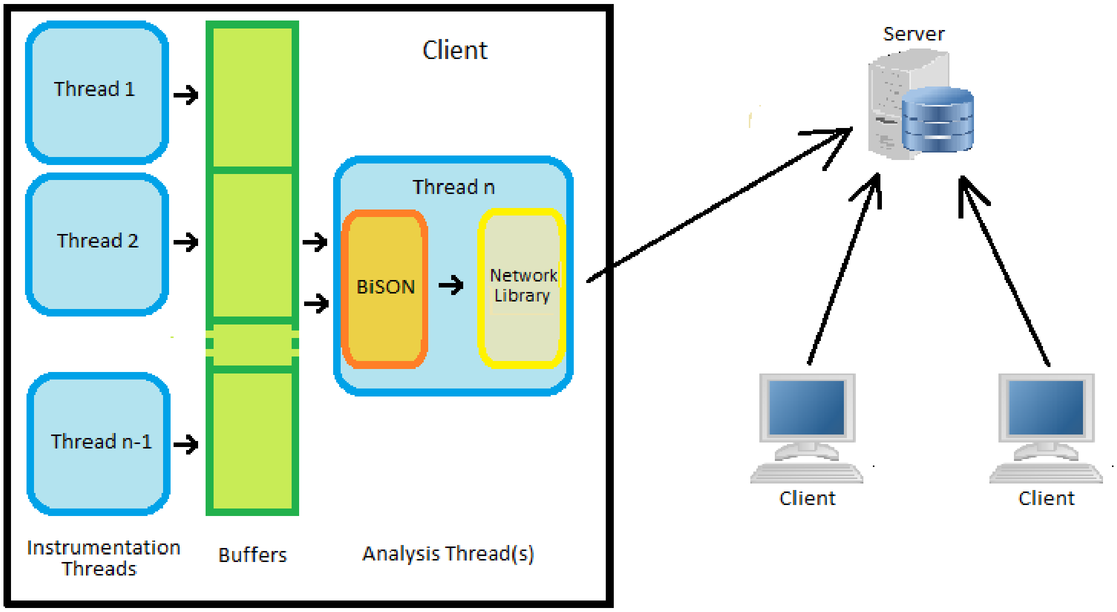

Figure 4.1.: An overview of the system. Arrows indicate the flow of data.

and marshalling libraries. The networking and marshalling libraries were designed to be user-friendly, fast, and for the marshalling library, easily extensible. The Binary JavaScript Object Notation (BSON) implementation was selected for the marshalling library because it offers a good combination of user readability and parsing speed, with a large variety of supported datatypes by default. Using this approach, it is hoped that analysis time can be reduced by offloading any heavy analysis routines to another system. For example, this approach would allow applications running on different machines to transmit instrumentation data to central location for storage and offline analysis.

4.3 Implementation of ReBIS

In this section, we will discuss various implementation details of ReBIS, including encountered issues and its solutions.

4.3.1 The Buffering System

The buffering system can be split into two parts: (1) the interface it exposes to the instrumentation and analysis threads; and (2) the internal workings of the system. The interface exposed to the instrumentation threads can be summed up in two functions. Before an instrumentation thread can write to the buffers, it must first create a buffer that the buffering system will associate with that particular thread by calling the *create_buffer* method. This ensures all instrumentation threads can write to the buffers at the same time without utilizing locks.

Algorithm 1 outlines the code executed during a write to buffer operation. It should be noted that a double buffer was utilized. As described in Algorithm 1, first the data is written to the buffer, and the position in the buffer incremented to the next available location. When the buffer is full, it is swapped with the second buffer. This allows for continual writing to the buffer while the other buffer is waiting to be consumed. The full buffer is then consumed by the analysis thread, which runs the pseudocode found in Algorithm 2.

Algorithm 1 Pseudocode for writing to a buffer.

```
write data to next available location
if main_buffer_full then
    while backup_buffer_full do
        yield current thread
    end while
    swap main_buffer with backup_buffer
    store number of written elements in buffer
    backup_buffer_full = TRUE
end if
```

The interface exposed to the analysis threads is simpler, consisting of only one method. This method is a *register* method that the analysis thread calls to register itself as an analysis thread. The analysis thread then continuously loops through the buffers and passes full buffers to a registered consumer. Access to this loop is prevented by first having to acquire a readlock on line 2 of Algorithm 2. This allows multiple consumers to operate concurrently, and prevents contention when a new buffer is being created, which acquires the associated writelock. The double-checked locking pattern [23] on lines 4-6 is used on a lock associated with each individual buffer to remove the possibility of multiple threads consuming the same buffer at the same time, and improves runtime by only performing a check when the buffer is not full, rather than acquiring a lock. Finally, lines 14-21 ensure that when the application is terminating, all remaining buffers are consumed before the analysis thread can terminate.

Algorithm 2 Pseudocode for analysis threads

```
1:  while $TRUE$ do
2:      acquire global_buffers_readlock
3:      for double_buffer in buffers do
4:          if backup_buffer$isfull$ then
5:              acquire double_buffer lock
6:              if backup_buffer$isfull$ then
7:                  consume backup_buffer
8:                  backup_buffer_full = $FALSE$
9:              end if
10:             release double_buffer lock
11:         end if
12:     end for
13:     release global_buffers_readlock
14:     if Pin::is_process_exiting then
15:         for double_buffer in buffers do
16:             if backup_buffer$isfull$ then
17:                 goto 1
18:             else return
19:             end if
20:         end for
21:     end if
22: end while
```

4.4 The Networking Library

The purpose of designing a networking library was to ensure that a user could easily send packets over a network while running in the Pin environment. Initial attempts included trying to leverage the ACE networking library. As discussed in Section 4.1, porting ACE to Pin was hard, and created unresolvable problems. The networking library is based on the acceptor-connector [24] pattern, and is designed to be as simple as possible. For example, users are able to obtain a connected socket in about 5 lines of code.

The networking library uses the BSON standard as its protocol. We selected BSON because it allows data transmitted over the network to be stored directly into a database like MongoDB [25] without any additional unmarshalling. This will improve performance receiving data since MongoDB is designed to efficiently handle writing data [26]. Our BSON implementation was created to be compatible with Pin. It also allows for fast and easy data modification, and is guaranteed to work with our networking library in a Pintool.

When compared to other BSON implementations, our library allows users to add custom data types that can be marshalled and unmarshalled like the data types native to the BSON standard. In order to do so, the user must first implement our `Value` interface, shown in Listing 4.2, for their new type. An important note, the write and read methods must be able to write and read their value to a buffer. Also, the *type* method should return a unique `BSON_Type`, which is just a character. Next, the user must extend the `Value_Factory` class to support their custom type. Listing 4.1 shows the relevant value factory interface. To properly extend this class, the user must implement a new `Value * create_my_type (void)` function. Then, by inserting their unique `BSON_Type` and *create_my_type* method into the protected `map_`, their `Value_Factory` subclass is complete.

Once the custom `Value` and `Value_Factory` are implemented, the user integrates it into the framework by calling the static *set_instance* method before using any bison functionality, passing in their new class as a template parameter. This way, the user-defined `Value_Factory` subclass is used instead of the base class, and all subsequent interactions with the framework can utilize the new type.

```
class Value_Factory
{
public:
  template <class T>
  static void set_instance (void);

  Value * create_value (const char c);

  template <class T>
  T * create_value (void);
protected:
  typedef Value * (Value_Factory::*create_func) (void);
  typedef std::map <char, create_func> value_map;
  value_map map_;
};
```

Listing 4.1: Relevant Value_Factory interface

```
class Value
{
public:
  /// Destructor
  virtual ~Value (void);

  /// Write this Value to the writer
  virtual void write (BSON_Writer & writer) const = 0;

  /// Set this Value based on the contents of the reader
  virtual bool read (BSON_Reader & reader) = 0;

  /// Return a unique identifier for this Value
  virtual BSON_Type type (void) = 0;

protected:
  /// Default constructor
  Value (void);
};
```

Listing 4.2: Value interface

5 RESULTS OF REBIS

In this chapter, we will outline some of the major benchmarking and experimental results.

5.1 Generating Results

All results were generated using machines using an AMD Opteron 4130 2.6GHz quad-core processor running Ubuntu 12.04 LTS. For tests requiring networking capability, the machines were connected via virtual LAN with an average round trip time of 0.105ms. Pin 2.14 build 71313 was used.

5.1.1 BSON Benchmarking

To benchmark the BSON implementation, it was compared to two existing C++ BSON implementations that were found on the BSON website (bsonspec.org). The first implementation is the MongoDB driver [27], in particular their legacy driver. The legacy driver was chosen as it was tested to work on both Windows and Linux systems. The other implementation was created by Project Kenai [28], and also treats BSON documents as a collection of elements, as opposed to the MongoDB implementation that treats BSON documents strictly as an array of characters. Two different optimizations were added to BiSON, so to properly benchmark them, a baseline without any optimizations was made, followed by both optimizations separately, then both optimizations together. The optimizations were a memory pool (mempool) and a no copy on write optimization (nocopy).

Four different types of tests were run to fully benchmark BiSON, which were write, read, search and delete. Write tests benchmarked the amount of time it took to add various values to a BSON document, and then obtain a character array representing the final written product. Read tests benchmarked the amount of time it took to generate an implementation-specific document object given an encoded-BSON document. Since the MongoDB implementation treats BSON objects as an array of bytes, no read tests were run on it. Search tests benchmarked the amount of time it took to access every object in a document. This would allow us to obtain an average access time for an element, if one were to access an element at random. Finally, delete tests benchmarked the amount of time it took to remove each element from a document one at a time.

For each test type, a set of tests were generated for each non-deprecated BSON element type. The number of elements tested were 1, 10, 100 and 1000, and for elements with variable length, such as strings, a length of 100 was used. Special case tests were given to an ”all” test, which tested all BSON elements except for arrays and documents, an array/document test which created an inner array/document into which was inserted every other BSON element except for arrays and documents, an empty array/document test, and a nested array/document test, where every array/document was inserted into the previously inserted array/document.

A note must be made here that the Project Kenai implementation did not implement the ability to insert maxkey and minkey types.

5.1.2 Networking

Benchmarking the networking library involved comparing the runtimes of various Pintools, with those same Pintools after adding a send of the data to the respective analysis routines, *i.e,* after incrementing the count, a copy of the new count was sent to another machine. Each Pintool was run 5 times, and an average taken. The number of messages received by the server was also recorded.

5.1.3 ReBIS

These tests were performed on the same program used in the networking experiments, except with varying the number of threads created, but with each thread performing the same amount of work. Two sets of tests were performed for each test case, one where no CPU binding occurred, and one where the instrumentation threads were bound to 3 cores, and the analysis thread bound to the last core. In addition to varying the number of threads, the size of the buffer was also varied, in order to see its effect on the execution time.

5.2 BSON

This section discusses the results of our BSON benchmarking tests. The graphs displayed in this section should be read as follows: from front to back, we have 1, 10, 100 and 1000 elements; and from left to right in each cluster of bars, we have baseline, mempool, nocopy, mempool + nocopy, Project Kenai, and, if the sixth bar exists, MongoDB implementations.

5.2.1 BSON Document Marshalling

Figure 5.1 shows the results from our BSON marshalling tests. In all of the optimization tests, adding the nocopy functionality decreased the time to obtain a character string representing the BSON document; nocopy was faster than the baseline, and mempool + nocopy was faster than mempool. The results show that for the simple types whose lengths are fixed, including the empty and nested array and document tests, for element counts of 1, 10 and 100, nocopy obtained the fastest results, with mempool producing the slowest results. This changed for the 1000 element count, where mempool + nocopy obtained the fastest result, while the baseline was the slowest. This was likely due to the time saved from allocating chunks of memory at once, rather than one element at a time.

Compared to the Project Kenai implementation, all BiSON implementations almost outperformed it in all counts, with only a couple of tests being even or slightly slower at counts of 1 and 10. This suggests that in normal usage, BiSON will likely outperform the Project Kenai implementation, with the difference increasing as more elements are used.

As expected, MongoDB was the fastest implementation,in large due to its treatment of BSON documents as an array of bytes, rather than like our treatment of BSON documents as collections as elements. This allows the MongoDB implementation to skip our intermediate step of inserting elements into a document before writing them to the buffer.

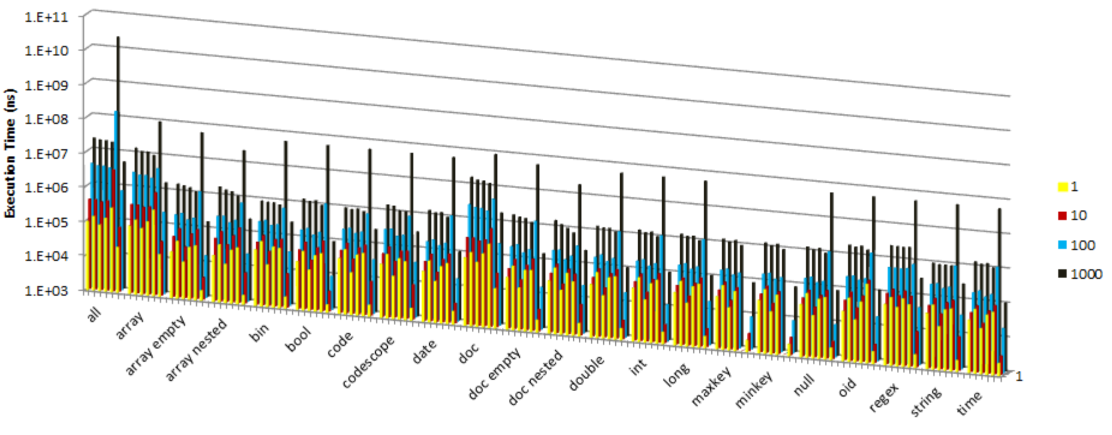

Figure 5.1.: Performance results for writing various datatypes to a BSON-encoded document.

5.2.2 BSON Element Searching

Figure 5.2 shows the results of our element searching tests. All of the BiSON optimizations and baseline performed similarily, with only a few % difference between then. Not surprisingly, MongoDB performed the worst in these tests. Our storage of elements in an intermediate data structure enables much more efficient search algorithms compared to MongoDB's linear search. However, our implementations were slightly slower than Project Kenai's at greater element counts.

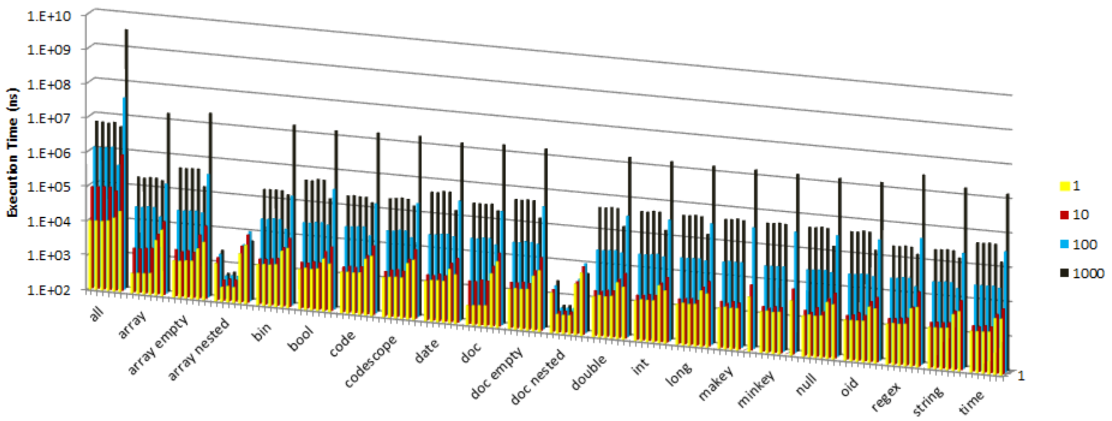

Figure 5.2.: Performance results for searching a BSON document.

5.2.3 BSON Document Demarshalling

Figure 5.3 shows the results of our BSON demarshalling tests. Unfortunately, many of the Project Kenai tests here failed, by throwing a segmentation fault, or otherwise giving inaccurate results, making comparisons for counts of 10, 100 and 1000 largely useless. When the Project Kenai implementation produced usable results, they were between 2X and 100X slower than their mempool + nocopy counterpart. Overall, our mempool implementations performed the best, due to the ability to allocate large chunks of memory at once, rather than one element at a time.

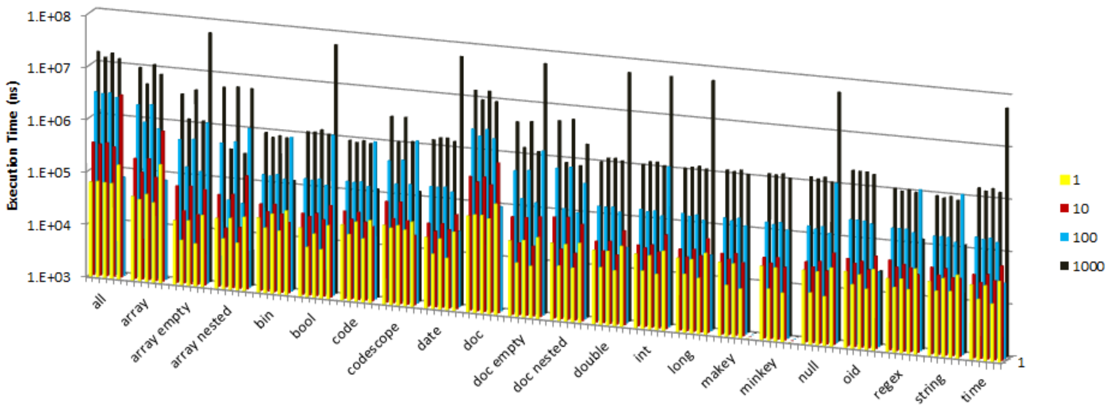

Figure 5.3.: Performance results for reading a BSON-encoded document containing various datatypes, and create an implementation specific BSON document.

5.2.4 BSON Element Deletion

Figure 5.4 shows the results of our element deletion tests. Of the BiSON optimizations, the mempool implementations performed slightly worse than without the memory pool. This suggests that our code for recycling the object's memory is slower than that to simply delete the object. Compared to the Project Kenai implementation, mempool + nocopy varies from being about the same speed for 1 and 10 elements, to being hundreds of times faster for 1000 elements. The MongoDB implementation was the fastest for 1 element, but slows down as more elements were deleted, becoming hundreds of times slower for 1000 elements. This was due to MongoDB having an extremely inefficient deletion routine, having to rewrite the entire document minus that element. Interestingly, the Project Kenai implementation was the slowest. In addition, it was nonintuitively difficult to delete all elements from a document; it had to be done in reverse order, or else their indexing scheme would break.

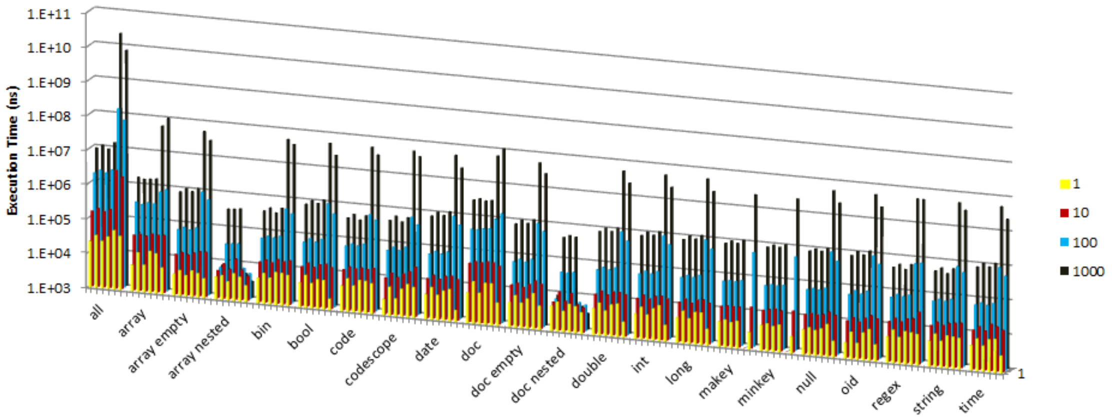

Figure 5.4.: Performance results for deleting various datatypes from a BSON document.

5.2.5 Ease of use

Overall, all libraries had a similar ease of use, except for the issue with deleting elements in the Project Kenai implementation. However, our implementation does provide some extra benefits compared to the MongoDB implementation. With the MongoDB implementation, in order to create an inner document/array, the most efficient method is to call a `subobjStart` or similar method, which returns another builder to be used for writing to that inner document/array. Some interesting issues arose from this. First, the user must keep track of which builder is which as created documents/arrays must be closed in the correct order, since incorrect closure will result in parsing issues. Secondly, the user could also accidentally forget to close an inner document/array, which would distort the layout of the documents/arrays. Lastly, there is nothing stopping a user from writing to a builder from a previous level while it is not the current 'active' builder. While BiSON is slower in many cases, it does provide a higher level of abstraction, preventing the user from having to deal with these lower level issues. In fact, BiSON does not require users to explicitly close any documents/arrays, that is automatically done whenever it is required. In addition, so long as simultaneous write to a single document/array is not done, a user could simultaneously write to separate documents/arrays that are contained within the same document/array. This fact can lead to situations where writing to a BiSON document is much faster than to a MongoDB document.

5.3 Networking

As seen in Figure 5.5, the increase in overhead varies greatly between Pintools. The main cause of this variation is due to how intrusive the Pintool is. Inscount0 was the most intrusive, performing instrumentation at an instruction level, and as a result, resulted in the largest amount of overhead. Compare this to the other inscount Pintools, which operate at a trace level, and had much lower overhead, and the malloc count Pintools, which operate at a routine level, and had even lower overhead. However, this does not paint the whole picture. Figure 5.6 shows that an increase in the number of messages sent causes an increase in the overhead, and that the relationship is fairly linear. However, the proccount Pintool seems to be an outlier, requiring almost half the time expected to execute. Including all of the points, a linear regression returned an R-squared value of 0.85. If we treat the proccount Pintool as an outlier, and plot a new linear regression line, we now obtain an R-squared value of 0.99. This suggests that users can use the amount of messages they plan on sending/receiving to estimate the overhead of their tool.

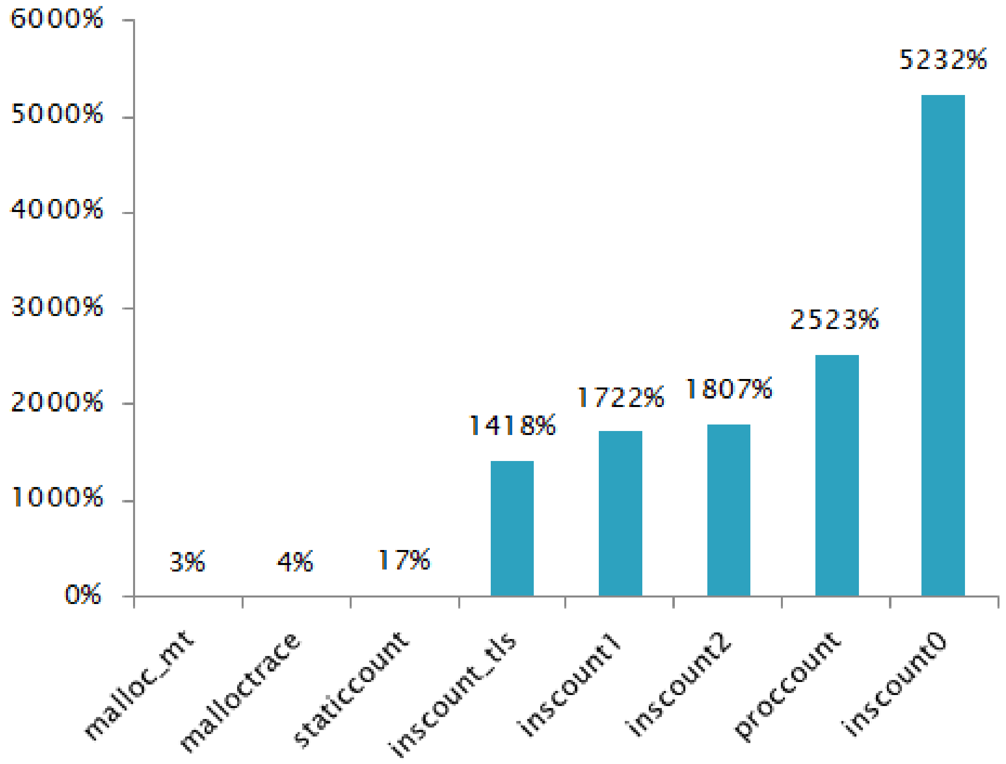

Figure 5.5.: Graph showing the increase in overhead caused by adding a send of the data in every analysis routine.

5.4 ReBIS

Table 5.1 shows that given a large enough buffer, the system can perform better without CPU binding than with CPU binding. However, as the table shows, an increasingly larger buffer is needed as the application spawns more threads to maintain this improved performance. What isn't shown, is that buffer size has little effect on the runtime of the system with CPU binding. This is more important in situations where memory is limited, such as in embedded systems, or where the application is

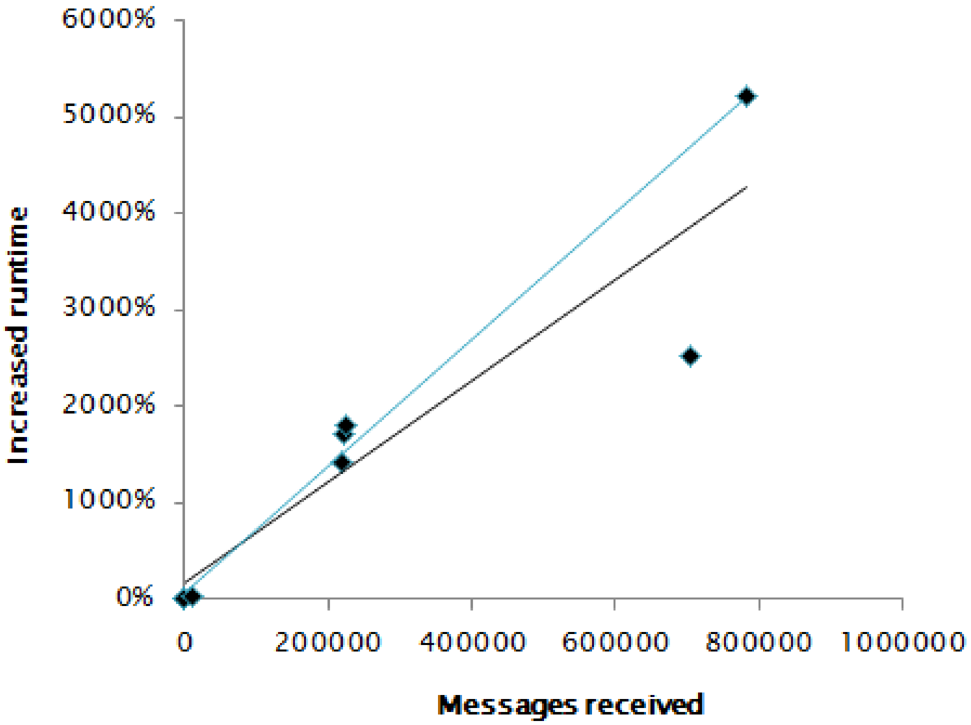

Figure 5.6.: Graph showing the effect of increased messages sent vs the increase in overhead.

extremely memory-intensive. The drastic change as you go down the table is caused almost entirely by variations in the no binding tests.

Comparing the execution times of our system in Table 5.1 with the more naive inscount0 implementation shown in figure 5.5 (actual runtimes not shown), our system managed to improve the runtime by about 1098% for a program that spawned 20 threads. For perspective, this was only about 486% slower than a vanilla inscount0 Pintool written using Pin++ that only increments a count, with no data collection or sending of data.

Table 5.1.: % diff. runtimes of ReBIS with vs. without binding of instrumentation and analysis threads to separate CPU cores.

Buffer size	Number of threads			
	8	20	40	240
50	-7.990	-24.055	-43.064	-74.041
100	-1.743	-11.003	-18.324	-59.521
250	0.836	5.802	2.469	-27.793
500	4.054	9.009	12.956	-2.597

6 CONCLUDING REMARKS

Dynamic binary instrumentation (DBI) is a powerful concept that enables users to obtain detailed information about a running program without modifying the program's original source code. The main issue, however, with DBI is the overhead associated with the non-intrusive instrumentation approach. Prior research techniques have been proposed to lessen the overhead, most of which use parallelization and buffering data.

In this thesis, a novel technique was introduced that incorporates data buffering, binding of instrumentation and analysis threads to separate CPU cores, and the introduction of a lightweight analysis routine replacement that offloads collected data over a network. Using our technique, we were able to create an inscount Pintool that ran around 11X faster than a naive inscount Pintool that sent data after every instruction, and was less than 5X slower than a native inscount Pintool that contains no analysis routine. Our framework was also found to require much less memory to achieve improved runtimes with CPU binding than without. We also learned that one analysis thread can handle at least three instrumentation threads when collecting data after every application instruction (*i.e.*, the use case that has the most overhead in DBI).

Based on our results and the functionality provided by ReBIS, here are potential future research directions:

- **Realtime Feedback** Using our framework, it may be possible to further reduce overhead by developing some form of feedback system. Using information collected on the application and the machine performing the instrumentation, it may be possible for the server to reconfigure the tool improve performance. Such reconfiguring could include spawning more analysis threads, modifying buffer sizes, or any other task to improve or modify instrumentation.

- **Distributed Instrumentation** Another future research direction is the exploration of how ReBIS can aid in overhead reduction via a distributed instrumentation system. In this context, each node instruments the same application in tandem, sending the data to a central server. These nodes would then be able to share the overhead by using some sampling algorithm, or having the server act as a scheduler to determine which part of the application each node instruments. This way, no single node instruments the entire application, but together, the nodes reach 100 % coverage.

- **Instrumentation of Distributed Systems** Using our framework, it may be possible to more easily instrument distributed systems. Given the nature of distributed systems, our inclusion of a networking library into our framework should allow for users to simultaneously instrument all nodes of a distributed system and send that data to a central processing server more easily. However, our framework is currently geared towards more general instrumentation needs. It may be possible to further extend it to more easily fit the needs of the distributed environment.

ReBIS has been integrated into Pin++, is freely available in open-source format, and can be downloaded from https://github.com/SEDS/PinPP (accessed May 6, 2016).

REFERENCES

REFERENCES

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

[2] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James 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, pages 2–11, New York, NY, USA, 2010. ACM.

[3] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 147–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce 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*, pages 28–36, 2008.

[5] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, March 2010.

[6] Pranith D. Kumar, Anchal Nema, and Rajeev Kumar. Hybrid analysis of executables to detect security vulnerabilities: Security vulnerabilities. In *Proceedings of the 2Nd India Software Engineering Conference*, ISEC '09, pages 141–142, New York, NY, USA, 2009. ACM.

[7] 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. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[8] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.

[9] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[10] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[11] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 155–174, New York, NY, USA, 2009. ACM.

[12] 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.

[13] James H. Hill and Dennis C. Feiock. Pin++: An object-oriented framework for writing pintools. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 133–141, New York, NY, USA, 2014. ACM.

[14] Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 209–220, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Dan Upton, Kim Hazelwood, Robert Cohn, and Greg Lueck. Improving instrumentation speed via buffering. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 52–61, New York, NY, USA, 2009. ACM.

[16] Danilo Ansaloni, Walter Binder, Abbas Heydarnoori, and Lydia Y Chen. Deferred methods: accelerating dynamic program analysis on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 242–251. ACM, 2012.

[17] Pin 2.14 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html, (accessed May 6, 2016).

[18] PinPP Repository. https://github.com/SEDS/PinPP, (accessed May 6, 2016).

[19] BSON. http://bsonspec.org/, (accessed May 6, 2016).

[20] JSON. http://json.org/, (accessed May 6, 2016).

[21] OpenDDS. http://opendds.org/, (accessed May 6, 2016).

[22] Douglas C. Schmidt. The adaptive communication environment: An object-oriented network programming toolkit for developing communication software. pages 214–225, 1993.

[23] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[24] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[25] MongoDB. https://www.mongodb.com/, (accessed May 6, 2016).

[26] MongoDB architecture guide. http://s3.amazonaws.com/info-mongodb-com/MongoDB_Architecture_Guide.pdf, (accessed May 6, 2016).

[27] MongoDB mongo-cxx-driver. https://github.com/mongodb/mongo-cxx-driver/tree/legacy, (accessed May 6, 2016).

[28] Project Kenai BSON C++ API. https://kenai.com/projects/mongoviewer/pages/BSON, (accessed May 6, 2016).