Pin++: A Object-oriented Framework for Writing Pintools *

James H. Hill Dennis C. Feiock

Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
{hilljh, dfeiock}@iupui.edu

Abstract

This paper presents a framework named *Pin++*. Pin++ is an object-oriented framework that uses template metaprogramming to implement Pintools, which are analysis tools for the dynamic binary instrumentation tool named Pin. The goal of Pin++ is to simplify programming a Pintool and promote reuse of its components across different Pintools. Our results show that Pintools implemented using Pin++ can have a 54% reduction in complexity, increase its modularity, and up to 60% reduction in instrumentation overhead.

Keywords Pin, Pintools, template metaprogramming, framework

1. Introduction

Pin [1] is a *dynamic binary instrumentation (DBI)* [2] tool for the IA-32 and x86-64 instruction-set architecture. It enables the creation of dynamic program analysis tools called *Pintools*. Pintools have been created to analyze a wide variety of concerns within programs, such as program faults [3], program behavior [3–5], performance profiling [4, 5], and root-cause analysis [6]. Examples of other DBI tools include, but is not limited to: DynamoRIO [7], DynInst [8], Solaris Dynamic Tracing (DTrace) [9], and Valgrind [10].

When developers implement a Pintool, they use the C++ programming language. Although developers use C++, Pintools interface with Pin using a C-like *application programming interface (API)*. This does not imply developers are forced to implement their Pintools using C, but the current design and structure promoted by Pin through its interface result in developers creating C-like Pintools. For example, the examples discussed in Pin's user manual [11] and distributed with Pin are implemented as if it was a traditional C program (*i.e.*, C functions, global variables, and no information hiding).

Unfortunately, current techniques for implementing a Pintool can result in designs that are fragile [12] (*i.e.*, the tendency of the software to break in many places every time it is changed), rigid [12] (*i.e.*, the tendency for software to be difficult to change, even in simple ways), and have components that are *hard* to reuse due to high cyclomatic complexity [13] (*i.e.*, the number of linearly independent paths through a program's source code) and low modularity [14]. For example, if several components in a Pintool are dependent on one or more global variables, then it is hard to reuse the component across different Pintools due to this coupling. Likewise, changing the global variable's definition can break the design in many locations.

There are also inherent complexities that complicate the design and implementation of a Pintool. For example, developers have to manually validate that analysis routines are registered with Pin using a parameter list that matches the analysis routine's signature. This is because there is no check in place—at compile time or runtime—to validate this requirement (see Section 2 for an example).

Although Pin is widely used today, implementing a Pintool can be a complicated process. Given the capabilities of C++, such as abstraction, encapsulation, and template metaprogramming, many complexities in a Pintool should not exist. Most importantly, developers should be capable of implementing Pintools from "components engineered with reuse and for reuse in mind instead of reinvention" [15]. To achieve this desire, however, Pin needs a supporting framework that embodies these principles.

For these reasons we implemented Pin++, which is an object-oriented framework for implementing Pintools. Pin++ uses software design patterns [16] to promote reuse and reduce the complexity of a Pintool. It also uses template meta-programming [17, 18] to reduce potential development errors and optimize a Pintool's performance at compile-time. Lastly, Pin++ is engineered to promote reuse of different components in a Pintool; it codifies required functionality so

* This work was sponsored in part by the Australian Defense Science and Technology Organization (DSTO).

developers do not have to re-implement such functionality for each and every Pintool.

The main contributions of this paper are as follows:

- It highlights accidental and inherent complexities associated with implementing a traditional Pintool;
- It discusses how Pin++ addresses accidental and inherent complexities associated with implementing a traditional Pintool;
- It quantitatively evaluates Pin++'s impact on the complexity (a design metric), modularity (a design metric), and instrumentation overhead (a performance metric) of a Pintool;
- It provides suggestions on what features should be removed from Pin and placed in a supporting framework, such as Pin++.

We validated Pin++ by implementing examples distributed with Pin using Pin++. Our results show that Pin++ can reduce the complexity of a Pintool up to 54%, improve the modularity of a Pintool, and reduce instrumentation overhead by up to 60%. Each of these improvements, however, is dependent on what features of Pin are used.

Paper organization. The remainder of this paper is organized as follows: Section 2 introduces a simple example to illustrate the challenges associated with writing a Pintool; Section 3 discusses the design and implementation of Pin++; Section 4 presents results from comparing traditional Pintools against ones implemented using Pin++; Section 5 discusses different design decision's impact on performance; Section 6 compares our work to other related works; and Section 7 provides concluding remarks.

2. The Complexities of Implementing a Pintool

Listing 1 illustrates an example Pintool from Pin's user manual. It is actually the first example in Pin's user manual.

```
1  #include <iostream>
2  #include <fstream>
3  #include "pin.H"
4
5  ofstream OutFile;
6
7  // The running count of instructions is kept here
8  // make it static to help the compiler optimize docount
9  static UINT64 icount = 0;
10
11 // Function called before every instruction is executed
12 VOID docount() { icount++; }
13
14 // Pin calls this function every time a new instruction
15 // is encountered
16 VOID Instruction(INS ins, VOID *v) {
17   INS_InsertCall(ins, IPOINT_BEFORE,
18                  (AFUNPTR)docount, IARG_END);
19 }
20
21 KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
22                             "o", "inscount.out",
23                             "specify output file name");
24
25 // This function is called when the application exits
26 VOID Fini(INT32 code, VOID *v) {
27   OutFile.setf(ios::showbase);
28   OutFile << "Count " << icount << endl;
29   OutFile.close();
30 }
31
32 INT32 Usage() {
33   cerr << "Counts number of dynamic instructions executed"
34        << endl << endl << KNOB_BASE::StringKnobSummary()
35        << endl;
36   return -1;
37 }
38
39 int main(int argc, char * argv[]) {
40   // Initialize pin
41   if (PIN_Init(argc, argv)) return Usage();
42
43   OutFile.open(KnobOutputFile.Value().c_str());
44
45   // Register instruments and callbacks
46   INS_AddInstrumentFunction(Instruction, 0);
47   PIN_AddFiniFunction(Fini, 0);
48
49   // Start the program, never returns
50   PIN_StartProgram();
51   return 0;
52 }
```

Listing 1. Pintool that counts the number of instructions executed by a program.

As shown in this example, there are several complexities associated with a traditional Pintool:

1. **Hard to see the design and structure of a Pintool.** A Pintool consists of several key components: a tool, one or more instruments, and one or more analysis routines. In Listing 1, the `docount` function (line 12) is the analysis component; the `Instruction` function (line 16) is the instrument component; and the remaining functions comprise the tool component. Although this is a small example, it is hard to see the individual components that constitute a Pintool. As the Pintool increases in size and complexity, it becomes *harder* to see the design and structure of a Pintool. This is an accidental complexity that can result in Pintools having rotten designs [12] (*e.g.*, fragility and rigidity due to the use of global variables); and exhibiting code smells [19] (*e.g.*, the use of global variables, too many parameters, and vertical separation) and antipatterns [20] (*e.g.*, spaghetti code, stovepipe Pintool, and reinvent the wheel).
2. **Hidden constraints between analysis routine definition and its registration with Pin.** In Listing 1, the developer creates an analysis routine (line 12). In order to register the analysis routine for callback, the developer has to first register it with Pin. As shown on line 17, the analysis routine is registered against each instruction executed by the program under instrumentation. As part of the registration process, the developer must remember the type of each parameter in the analysis routine's argument list, and specify it when invoking `INS_InsertCall`. The developer must also end the argument list specification with `IARG_END`. In this exam-

ple, since the docount function has no arguments, there are no parameters included with INS_InsertCall.

In some cases, there are analysis routine argument types that require an extra parameter. In this situation, it is the developer's responsibility to remember this requirement. Unfortunately, there are no mechanisms in Pin to address this complexity at compile-time (*i.e.*, validating that arguments used to register an analysis routine match its expected parameters, and validating that required extra arguments are provided when registering the analysis routine). Instead, the developer realizes the problem exists at runtime when the Pintool fails.

3. **No inherent reuse of Pintool components.** In the example from Listing 1, each function can be considered a component in the Pintool. Unfortunately, the Pintool's design hinders reuse of its components. This is because there are multiple global variables used by each of the components in the Pintool.

   For example, the docount function (or component) and the Fini component use the icount global variable. Because the docount component is dependent on a global variable, it is hard to reuse this simple analysis component in other Pintools. If developers wanted to reuse the component, then they would have to either declare a similar global variable in the Pintool, or understand how to pass stateful information to Pin.

4. **Reinvention of required behavior.** When implementing a Pintool, there are several required steps that must be implemented in all Pintools. First, the developer must correctly initialize the environment (line 41). Secondly, the developer must register each instrument, which is the component responsible for inserting analysis routines into the program (line 46). Lastly, the developer must start the program being instrumented (line 50).

   These steps must be completed in a specific order. For example, you cannot register any callbacks with Pin before initializing the environment. Although this seems minor, this code is re-invented across each tool. Unless there is a special use case for a Pintool, such as a static Pintool [11], then developers should not have to manually implement bootstrapping and initialization code for each Pintool. Moreover, it is hard to get reuse of components because each Pintool is reimplementing the same code. This is because developers will be responsible for understanding how to programmatically bootstrap the component instead of the component understanding how to bootstrap itself when included in the Pintool.

As discussed above, there are many complexities associated with writing a Pintool. Some of these complexities impact design metrics, such as cyclomatic complexity and modularity—making it hard to maintain a Pintool. Other complexities impact the performance of a Pintool, causing the Pintool to add more overhead than wanted to the program under instrumentation. The remainder of this paper will therefore discuss how Pin++ helps address the complexities discussed above—improving the design, implementation, and performance of a Pintool.

3. The Design and Implementation of Pin++

This section discusses how Pin++ address complexities introduced in Section 2.

3.1 Improving the Design and Structure of a Pintool

As shown in Listing 1 and discussed in Section 2, one of the complexities of implementing a Pintool is difficulty seeing its design and structure. We know that Pintools consist of several key components, but current designs can easily hide this structure behind a myriad of *spaghetti code* [20]. Pin++ addresses this complexity by requiring each key component in a Pintool be defined as an object. This approach helps promote better design based on accepted object-oriented design principles [12] and software design patterns [16].

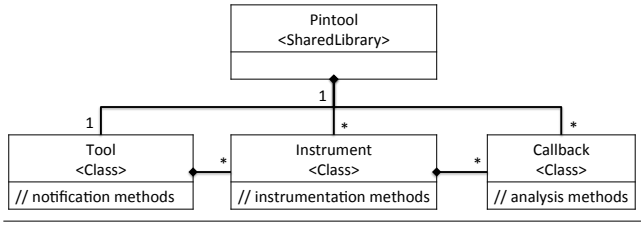

Figure 1. Composition of a Pintool in Pin++.

As shown in Figure 1, a Pintool in Pin++ consists of the following key components (or objects):

- **Tool** object is responsible for connecting Pin with the instrument objects;
- **Instrument** object is called by Pin when the Pintool needs to instrument the element of interest; and
- **Callback** object is called when the Pintool is to analyze data at an instrumentation point.

Listing 2 illustrates the example in Listing 1 implemented using Pin++. As shown in this listing, callbacks subclass from the Callback class in Pin++ (line 8). Pin++ uses the Curiously Recurring Template Pattern [21] to define what data is collected by the callback. The next section discusses this design choice in more detail. The callbacks must implement the handle_analyze() method (line 12). This method is invoked automatically by Pin, and is where the Pintool performs its analysis.

```
1  #include "pin++/Callback.h"
2  #include "pin++/Instruction_Instrument.h"
3  #include "pin++/Pintool.h"
4
5  #include <fstream>
6
7  // Pin++ callback object
8  class docount : public Callback < docount (void) > {
9  public:
10   docount (void) : count_ (0) { }
```

```
11
12    void handle_analyze (void) { ++ this->count_; }
13    UINT64 count (void) const { return this->count_; }
14
15  private:
16    UINT64 count_;
17  };
18
19  // Pin++ instruction-level instrument
20  class Instruction :
21    public Instruction_Instrument <Instruction> {
22  public:
23    void handle_instrument (const Ins & ins) {
24      this->callback_.insert (IPOINT_BEFORE, ins);
25    }
26
27    UINT64 count (void) const {return this->cnt_.count();}
28
29  private:
30    docount cnt_;
31  };
32
33  // one and only Pin++ tool object
34  class inscount : public Tool <inscount> {
35  public:
36    inscount (void) {
37      this->enable_fini_callback ();
38    }
39
40    void handle_fini (INT32 code) {
41      std::ofstream fout (outfile_.Value ().c_str ());
42      fout.setf (ios::showbase);
43      fout <<  "Count "
44           << this->instruction_.count () << std::endl;
45
46      fout.close ();
47    }
48
49  private:
50    // One and only instrument
51    Instruction instruction_;
52
53    /// @{ KNOBS
54    static KNOB <string> outfile_;
55    /// @}
56  };
57
58  KNOB <string> inscount::
59  outfile_ (KNOB_MODE_WRITEONCE, "pintool", "o",
60            "inscount.out", "output file name");
61
62  // macro simplifying pintool initialization
63  DECLARE_PINTOOL (inscount);
```

Listing 2. Pintool in Listing 1 re-implemented using Pin++.

Instrument components in Pin++ (line 21) subclass from the instrument class that corresponds to the desired level of instrumentation. Table 1 lists the different types of instrument classes available in Pin++, which correspond to those available when implementing a Pintool using the traditional method. As shown in Listing 2, each instrument component must implement the `handle_instrument()` method (line 23). This method is called automatically by Pin. Within the `handle_instrument()` method, the instrument is responsible for inserting callbacks into the program under instrumentation.

Lastly, the tool component in Listing 2 (line 34) is responsible for handling notifications from Pin by implementing methods of interest defined in the base class. If a notification is not implemented, then it is optimized out of the final binary. This is possible since the base class does not use any virtual methods to implement polymorphic behavior.

Scope	Pin++ Type
INS	`Instruction_Instrument`
TRACE	`Trace_Instrument`
RTN	`Routine_Instrument`
IMG	`Image_Instrument`

Table 1. Different instrument objects available in Pin++.

Instead, it uses the Curiously Recurring Template Pattern to gain access to the subclass. Finally, the developer must register the tool object for the correct notifications from Pin. This new design and structure of a Pintool using Pin++ helps improve its quality, reduce its complexity (see Section 4), and provides foundation for addressing other complexities discussed in Section 2.

3.2 Removing Hidden Constraints Between Analysis Routine and its Registration Process

As discussed in Section 2, the argument types passed to the `*_InsertCall` function must match the signature of the analysis routine. If incorrect arguments types are provided, then the Pintool will have runtime failures. This hidden constraint is an inherent complexity of implementing a Pintool.

You will notice, however, that the source code in Listing 2 does not have this same requirement when instrumenting the instruction. As shown on line 24 in Listing 2, the user only need to call `insert`—passing a location and target object to instrument. The Pin++ framework then automatically constructs the correct parameter list and invokes the native `*_InsertCall` function as done in Listing 1.

We are able to achieve this desired behavior because the developer must parameterize the base class with a function type. The return type of the function type is the name of the subclass. This allows the base class to directly call into the subclass without using virtual methods. The function type parameters are the arguments that the callback expects to receive from Pin. In Listing 2, the callback does not expect any data from Pin and defines its function type as `<docount(void)>`.

```
1   class Mem_Read :
2   public Callback <Mem_Read(ARG_INST_PTR, ARG_MEMORYOP_EA)> {
3   public:
4     Mem_Read (FILE * file) : file_ (file) { }
5
6     void handle_analyze (arg1_type ip, arg2_type addr) {
7       fprintf (this->file_, "%p: R %p\n", ip, addr);
8     }
9
10  private:
11    FILE * file_;
12  };
13
14  class Instrument :
15  public Instruction_Instrument <Instrument> {
16  public:
17    Instrument (FILE * file) : mr_ (file) { }
18
19    void handle_instrument (const Ins & ins) {
20      UINT32 operands = ins.memory_operand_count ();
21      IPOINT loc = IPOINT_BEFORE;
22
23      for (UINT32 op = 0; op < operands; ++ op)
24        if (ins.is_memory_operand_read (op))
```

```
25          this->mr_.insert_predicated (loc, ins, op);
26      }
27
28    private:
29      Mem_Read mr_;
30    };
```

Listing 3. Code snippet of a Pintool in Pin++ that has a callback object with multiple arguments.

Listing 3 shows a callback that receives multiple arguments from Pin. As shown in this example, the callback expects to receive two pieces of data (*i.e.*, the current instruction pointer and memory address) each time its analysis method is invoked. Pin++ defines arguments that correspond to the arguments types in Pin. Pin++ uses its own argument type system because it provides more control over its generative programming process. It also allows Pin++ to control what argument types a callback object supports. We discuss in Section 5 why Pin++ does not support all the argument types supported by Pin.

Callbacks with arguments that have additional parameters. In some cases a callback expects to receive data, but only providing the argument type is not enough information for Pin to know what data to pass to the callback. For example, the `IARG_MEMORYOP_EA` argument expects an extra argument (*i.e.*, the memory operation index). Without the additional argument, Pin does not know what memory operation index the callback wants.

Pin++ handles this use case by providing the additional parameters after the target object parameter to the `insert` method. As shown on line 25 in Listing 3, the additional argument required for the `ARG_MEMORYOP_EA` parameter is supplied after the target object parameter. Each additional argument is supplied in order based on the signature of the function pointer passed to the base class of the corresponding callback. Pin++ then constructs an argument list containing the additional arguments. Lastly, this design approach allows Pin++ to type check each additional argument, which is not possible when implementing a Pintool using the traditional approach.

Validating analysis routine parameter types. When implementing a Pintool using the traditional method, it is not possible to validate the `IARG_*` values passed to the `*_InsertCall` function match the parameter types expected by the analysis routine. This is because the analysis routine parameter for the traditional insert call function is a function pointer that takes variatric arguments. It is therefore impossible for the compiler to type check the argument types against the signature of the analysis routine.

Because the function type passed to the base class of a callback object in Pin++ contains the expected argument types, Pin++ is able to validate the `handle_analyze` method of a callback object has the correct parameter types. If the parameter types of the `handle_analyze` method are incorrect, the developer gets a compilation error that details the expected signature of the `handle_analyze` method. Lastly, line 6 in Listing 3 shows how Pin++ provides type definitions for each expected parameter, which helps reduce compile-time errors and makes Pin++'s design less rigid and fragile.

Conditional analysis. In some cases, developers do not want their analysis routines to execute each time an instrumented object (see Table 1) is executed. Pin supports this need using `*_InsertIfCall` and `*_InsertThenCall` functions. These functions behave similar to the standard `*_InsertCall` functions. The former function defines the analysis routine that determines when the analysis routine should execute. The latter function defines the analysis routine that is to be executed (*i.e.*, it is same analysis routine registered using the standard insert call function).

With conditional analysis, developers must manage the same complexities as the standard insert call function discussed above. In addition, developers must (1) remember that the location (*i.e.*, before, after, or anywhere) passed to the if-function, must also be passed to the then-function; and (2) remember to call the if-function before the then-function. Finally, it is *hard* to switch between standard analysis and conditional analysis since you must remove the standard insert call, and replace it with the if-then insert calls.

```
1   class countdown :
2   public Conditional_Callback <countdown(void)> {
3   public:
4     countdown (void) : c_ (1) { }
5
6     // required method
7     bool do_next (void) { return (-- this->c_ == 0); }
8     void reset_counter (INT32 c) { this->c_ = c; }
9
10  private:
11    INT32 c_;
12  };
13
14  // ...
15
16  class Instrument :
17  public Instruction_Instrument <Instrument> {
18  public:
19    Instrument (FILE * file) : print_ (file, count_) { }
20
21    void handle_instrument (const Ins & ins) {
22      this->print_[this->count_].insert (IPOINT_BEFORE, ins);
23    }
24
25  private:
26    countdown count_;  // Conditional callback
27    printip print_;    // Standard callback
28  };
```

Listing 4. Conditional analysis in Pin++.

Pin++ addresses this complexity by extending its callback architecture to support conditional callbacks. As shown in Listing 4, conditional callbacks are implemented in the same manner as the analysis objects. The main difference is that conditional callback objects must implement a method named `do_next` that has a boolean return value. When this method returns `true`, then the analysis routine registered via the then-function is executed.

Conditional analysis is enabled in Pin++ via an overloaded index operator on the callback object. As shown in Listing 4, the conditional callback object appears in the in-

dex operator before calling the insert method on the callback object. Pin++ uses this syntax because it appears like a guard in many state machine languages. The final result of this code is that Pin++ automatically calls the if-then insert functions. If the index operator is removed (*i.e.*, `[this-->count_]`) then Pin++ reverts to using the standard insert function. This appoach simplifies enabling conditional analysis, and allows us to create reusable conditional callback objects that implement different sampling strategizes.

3.3 Engineering Pintool Components for Reuse

As explained in Section 2, current design and implementation techniques make it *hard* to reuse components in different Pintools. For example, to reuse the `docount` analysis routine or the `Instruction` instrument from Listing 1, the developer has to redefine state (*e.g.*, global variables) that should be managed by the most appropriate component.

Because Pin++ uses objects to define each of its components, its components support engineering for reuse. For example, it is possible to package each component in a library and then assemble the Pintool from the reusable components. Each component can also expose configuration methods for its altering behavior. For example, Listing 5 illustrates a simple batch counter callback that can be configured as needed using the `batch_count` method. This callback is included with the Pin++ framework for reuse in any Pintool.

```
1  class Batch_Counter :
2  public Callback <Batch_Counter(void)> {
3  public:
4    Batch_Counter (void) : c_ (0), bc_ (1) { }
5
6    void handle_analyze (void) {
7      this->c_ += this->bc_;
8    }
9
10   void batch_count (UINT64 c) { this->bc_ = c; }
11   UINT64 batch_count (void) const { return this->bc_; }
12   UINT64 count (void) const { return this->c_; }
13
14 private:
15   UINT64 c_, bc_;
16 };
```

Listing 5. Callback object with a configuration parameter.

Each component in Pin++ is designed to be a self-contained object composed from other components. In the case of a tool, it is composed from a set of instruments. In the case of an instrument, it is composed from a set of callback objects. Finally, each child object informs its parent (or container) object what it needs in order to function correct. For example, the `Memory_Read` callback in Listing 3 informs its container instrument that it requires a file handle.

This design approach allows developers to engineer each component in a Pintool for reuse. Moreover, this design methodology helps lay a foundation for creating a repository of components (*i.e.*, callbacks, instruments, and tools) that can be reused across different Pintools, which is *hard* to do with Pin right now.

3.4 Reducing Reinvention of Required Behavior

As discussed in Section 2, developers of Pintools are required to bootstrap each Pintool with Pin. This includes initializing the Pintool, registering the necessary callbacks, and starting the program under instrumentation. Some parts of the bootstrapping process is common across different Pintools, such as initialization and starting the program under instrumentation. Other parts of the bootstrapping process are not common across different Pintools, such as registering the different callbacks.

Pin++ addresses complexities associated with reinventing required behavior in two ways. First, Pin++ uses macros to define required behavior in a Pintool. As shown on line 63 in Listing 2, Pin++ uses the `DECLARE_PINTOOL(tool)` macro to bootstrap a Pintool. The macro has common behavior that must be implemented in a Pintool, such as defining the `main` function, initializing the environment, allocating to concrete tool component, and starting the program under instrumentation. This version of the macro starts the program in JIT mode. Pin++ also provides an equivalent macro to start a program in probed mode.

The second way Pin++ reduces reinvention of required behavior is in the registration process of components. In a traditional Pintool, the developer must manually register each instrument with Pin. As shown on line 46 in Listing 1, the developer has to use the `INS_AddInstrumentFunction` function to register the `Instruction` instrument.

Pin++ improves this aspect of a Pintool by leveraging semantics of a composing a Pintool in Pin++. As seen in Listing 2, there is no explicit call to a function that registers an instrument. This is because the instrument objects (see Table 1) contain this required behavior in its constructor. Pin++ only requires the developer instantiate an instrument to register it with Pin. This is the reason the developer only need to include the instrument object in a tool object, and the Pintool still functions correctly.

The developer can also lazily load instruments by not allocating the instrument object until it is needed. This allows developers to load instruments after the program has started. Destroying an instrument, however, does not unregister the instrument with Pin because Pin does not support such functionality for individual instruments.

Finally, the design of instruments in Pin++ assists with engineering an instrument component for reuse. This is because the instrument object is self-contained. It contains all the necessary logic for registering itself correctly when included in any Pintool. The developer therefore does not have to worry about how to register the instrument component in their tool. Instead, the developer focuses on configuring the instrument component using exposed configuration methods.

4. Experimental Results of Pin++

We evaluated Pin++ by implementing 21 different Pintools distributed with Pin using Pin++. For this evaluation, we were interested in answering two main questions:

1. **Does Pin++ reduce the complexity of a Pintool?** We are interested in answering this question because it will give us a better understanding of reuse, maintainability, and modularity of the code implemented using Pin++. Section 4.1 discusses our results to this question.
2. **Does Pin++ add additional overhead of the instrumentation process?** We are interested in answering this question because Pin++ is a layer of abstraction that resides between Pin and what the developer actually implements. When additional layers of abstraction are introduced into the software stack, such as with middleware [22], performance can be impacted negatively. We therefore want to make sure that Pin++ does not have too much negative impact on existing instrumentation overhead. Section 4.2 discusses our results to this question.

4.1 Question 1. Understanding Complexity When Using Pin++

We used CCCC (`cccc.sourceforge.net`) to collect metrics about the source code for the Pintools we used in our experiments. Figure 2 shows the cyclomatic complexity for the traditional Pintool implementation, and its equivalent implemented using Pin++. As shown in this graph, the cyclomatic complexity of a Pintool implemented using Pin++ is less. This is because there is less coupling between individual components in a Pintool when implemented using Pin++. For example, the Pintools implemented using Pin++ do not use any global variables; whereas, those implemented as a traditional Pintool make heavy use of global variables, which increases their cyclomatic complexity.

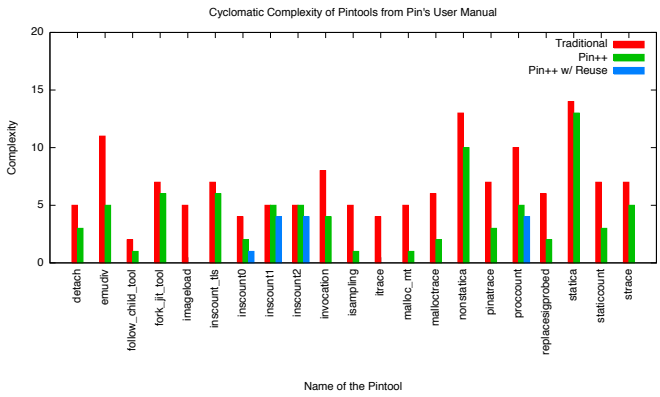

Figure 2. Cyclomatic complexity of a Pintool when implemented using the traditional versus Pin++ approach.

The cyclomatic complexity for `inscount{1|2}` is the same for both the traditional and Pin++ implementation. This is because the Pin++ implementation stores count information in each allocated callback object. In the finalize method, the Pintool must sum the count stored in each callback, which requires using nested for-loops. The traditional implementation updates a global counter, and therefore does not require any looping to produce the final count. When we redesigned the Pin++ implementation to remove the nested for-loop, its cyclomatic complexity went down to 1 (not shown in any graph). Its instrumentation overhead, however, increased upwards of 33% due to a large amount of indirect calls to update a referenced counter. We therefore reverted the Pin++ implementation of `inscount{1|2}` back to the one currently represented in Figure 2.

Last, we would like to point out that when we implemented the Pintool from reusable components, its complexity went down below that of the regular Pin++ implementation. As shown in Figure 2, we were only able to implement `inscount{0|1|2}` and `proccont` using a reusable callback object provided with the Pin++ framework because they were each variations of the same tool that relied on a counter-based analysis routine.

On SLOC. Although Pin++ reduces the cyclomatic complexity of a Pintool, the number of source lines of code (SLOC) of a Pintool increases when using Pin++. For example, Figure 3 shows the SLOC for each of the Pintools implemented using the traditional approach versus using Pin++. We expected the number of SLOC to increase because each component in Pin++ is implemented as a class instead of a function. This design choice in itself automatically increases the number of SLOC for a Pintool. When we implemented the Pintool from reusable components, the SLOC decreased close to the traditional implementation of the Pintool as expected.

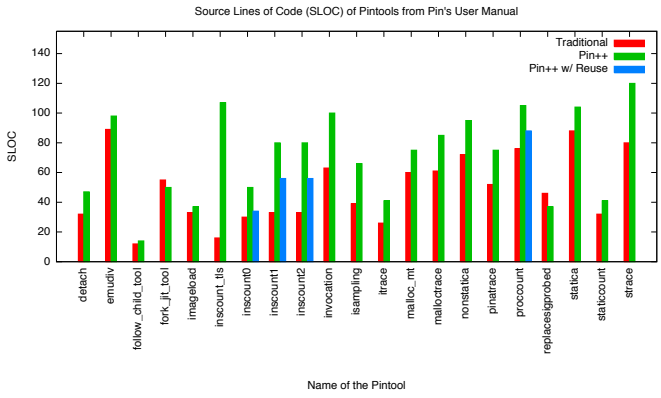

Figure 3. Source lines of code (SLOC) of a Pintool when implemented using the traditional versus Pin++ approach.

On modularity. One of the design goals of Pin++ was to engineer components for reuse. An indicator of reuse is the modularity of a program [14]. Figure 4 shows the modularity of a traditional Pintool when compared to its corresponding Pin++ implementation. As shown in this figure, Pintools implemented using Pin++ have a high modularity value when

compared to its traditional implementation. This means that components in a Pintool implemented using Pin++ are more capable of reuse when compared to its traditional implementation. Finally, the modularity of the Pintool decreases when constructed from reusable components because part of the Pintool that helped increase modularity is removed.

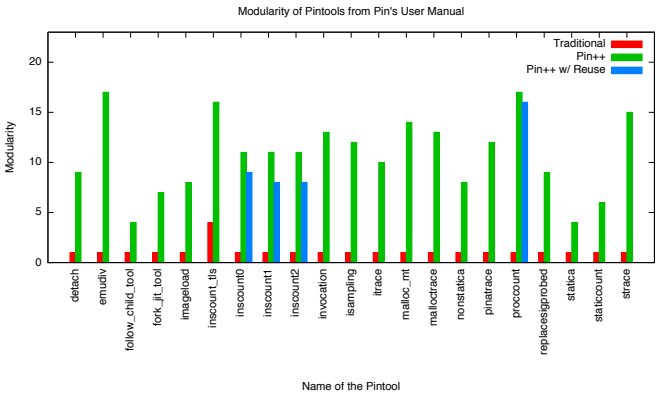

Figure 4. Modularity of a Pintool when implemented using traditional versus Pin++ approach.

4.2 Question 2. Understanding Performance When Using Pin++

We evaluated the performance of the traditional versus Pin++ implementation of the Pintools using the SPEC CPU2006 benchmark suite. Our performance tests were executed on Dell PowerEdge R815 with 2x AMD Operton 6272 (64-cores), 2.1 GHz processor, 32 GB RAM, and running Ubuntu 12.10. Each benchmark was compiled using GCC 4.7.2, with no changes to the SPEC CPU2006 compiler and linker flags. We executed each implementation of the Pintool (*i.e.*, traditional and Pin++) for 5 iterations where 1 iteration is the execution of both implementations. We did not evaluate the native benchmark execution times because the purpose of our experiments is to compare performance between the traditional and Pin++ implementation, and comparisons of native versus Pin performance already exist [1].

Figure 5 and Figure 6 show the results of our performance tests. If the bar is above 0%, then it is a reduction in instrumentation overhead. If the bar is below 0%, then it is an increase in instrumentation overhead. We used a subset of the Pintools from the previous section because the Pintools in Figure 5 and Figure 6 are more intrusive and not just showcasing simple features of Pin, such as how to detach from a program under instrumentation. Also, the Pintools in the figures above are the same ones used by the authors of Pin in previous performance tests [1].

As you can see in both figures, the Pin++ implementation for many of the Pintools and benchmarks reduces instrumentation overhead. Of all the Pintools we evaluated, `inscount_tls`, which uses thread-local storage to save instrumentation data for each thread in the program, has the

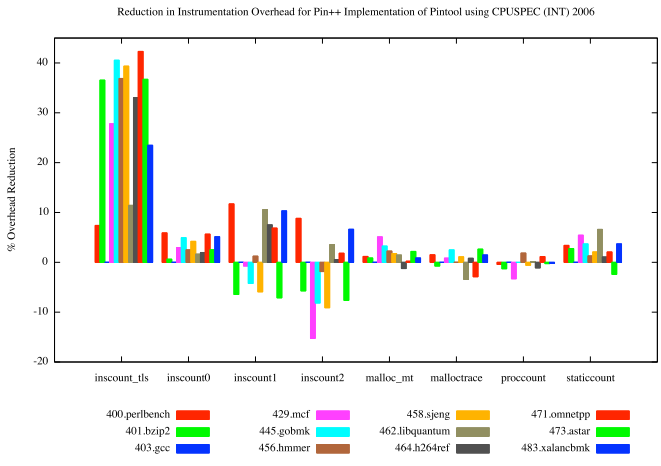

Figure 5. Percent difference in instrumentation overhead of traditional vs. Pin++ Pintool against integer benchmarks in SPEC CPU2006.

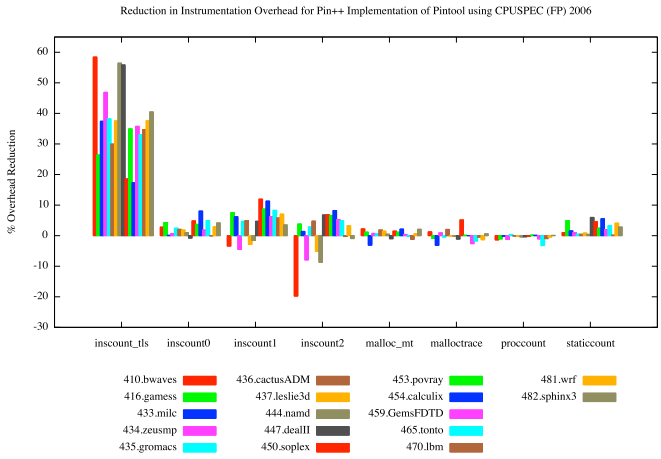

Figure 6. Percent difference in instrumentation overhead of traditional vs. Pin++ Pintool against floating point benchmarks in SPEC CPU2006.

most reduction in instrumentation overhead. `inscount0` and `staticcount` are the two most intrusive Pintools because they count every single instruction executed either dynamically at runtime or statically without executing the program, respectively. In both cases, the Pin++ implementation reduces the instrumentation overhead.

`inscount{1|2}` are the two Pintools where the Pin++ implementation added more instrumentation overhead in some cases. The traditional implementation of `inscount1` is an improvement to `inscount0` because it does not count every instruction, but performs a batch count of the number of instructions in a basic block (BBL)—a sequence of instructions with single entry and exit. `inscount2` is an improvement to `inscount1` because it uses fast calling mechanisms in Pin. The Pin++ implementation differs for

these two Pintools in that (1) it *always* uses fast call mechanisms in Pin; and (2) it allocates a new callback object for each BBL and stores the batch value in the callback object instead of registering the batch value with the analysis routine.

Our tests indicate that Pin++ performance benefit over Pin is more significant when Pintools are more intrusive (*e.g.*, `inscount_tls`, `inscount0`, and `staticcount`). For Pintools that are not as intrusive, there is less reduction in instrumentation overhead, and in some cases an increase in instrumentation overhead. Moving forward, one of our goals is to understand if we can enhance Pin++ to ensure that it does not add any extra overhead when compared to its traditional implementation for less intrusive Pintools.

5. Design Decisions Impact on Performance

The performance results in Section 4 show that Pin++ was able to reduce instrumentation overhead of a Pintool when compared to the traditional implementation. Although the results show this is the case, it is counterintuitive to what was expected. This is because Pin++ is an added layer of abstraction between Pin and the actual Pintool's application logic. It could therefore be expected that Pin++ will add more overhead of the Pintool. Our results, however, show this is not the case.

We did notice for some test runs that the Pin++ implementation performed worse than the traditional implementation, but the average performance for Pin++ was better. We therefore cannot definitively state that a Pintool written using Pin++ will *always* reduce instrumentation overhead since it depends on what features of Pin used. Lastly, the following design decisions helped reduce instrumentation overhead:

- **Remove registration of multiple global callbacks.** There are aspects of Pin where the developer can register multiple global callbacks. For example, developers can register multiple global callbacks for program lifecycle events, thread lifecycle events, and exceptions to name a few. It is understandable why such a feature is necessary, but it should not be implemented at the DBI level. Instead, the DBI tool should support a single callback. The single callback, which resides at the domain-specific level, is a bridge to registering multiple callbacks.

  Using this design approach should allow Pin to further optimize its design and performance. This is because Pin will not be concerned with managing and looping through multiple callbacks, which adds unwanted overhead to instrumentation. By moving such concerns into a supporting framework, such as Pin++, instrumentation overhead can be optimized case by case. For example, a Pintool that represents the minimal case (*e.g.*, one callback) will not suffer because Pin is implemented to support the broadest case (*e.g.*, multiple callbacks).

- **Prohibit registration of domain-specific data.** Pin allows a Pintool to register domain-specific data to be passed to the analysis routine. The parameters that allow domain-specific data include: `IARG_BOOL`, `IARG_ADDRINT`, `IARG_PTR`, and `IARG_UINT32`. For example, `inscount{1|2}` optimize the implementation of `inscount0` by counting the number of instructions in a BBL, and passing that number to the analysis routine each time a BBL executes.

  As shown in our results above, the Pin++ implementation of `inscount{1|2}` performs better (in majority cases) because Pin++ does not, and does not need to, support domain-specific data arguments. We therefore believe that Pin should remove such functionality from its design and let such data be managed by a framework, such as Pin++, where its access can be optimized accordingly. Likewise, the data remains close to where it is used [19].

6. Related Works

A majority of the inspiration for Pin++ comes from working with the Boost.Spirit [23] library. Boost.Spirit is a C++ library that uses template metaprogramming facilities to create parsers for context-free grammars. By using template metaprogramming facilities, Boost.Spirit parsers are implemented using a syntax that closely resembles *Extended Backus-Naur Form (EBNF)* [24]. This means that developers are implementing parsers using a domain-specific syntax that resembles how context-free grammars are described. Although Pin++ does not have a domain-specific language, its use of template metaprogramming to reduce the complexity of a Pintool is similar to how template metaprogramming helped the Boost.Spirit library reduce complexities associated with creating parsers for grammars in C++.

The Adaptive Communication Environment (ACE) [25] is an object-oriented framework that implements patterns for concurrent communication software. It provides reusable C++ wrapper facades [16] and framework components that perform common communication software tasks across a range of OS platforms. Our work relates to ACE because our decision to use the *resource allocation is initialization* idiom to manage the behavior of Pin is similar features provided by ACE. Likewise, Pin++ use facades to improve the usage of low-level C functions as done in ACE, and many other frameworks.

DynamoRIO [7], DynInst [8], and Valgrind [10] are DBI frameworks that allow developers to write analysis tools in C/C++. Although each DBI framework allows developers to write analysis tools in C++, many of the analysis tools use a mixture of C and C++. Similar to the traditional method of implementing a Pintool, this results in code that is unnecessarily complex. Moreover, it is hard to understand the structure and design of the analysis tools. We therefore believe that (1) these tool developers can learn from our experiences

with Pin++ and (2) the tools can benefit from layer of abstraction that resides between the DBI framework and the analysis tool with the goal of reducing accidental and inherent design complexities without impacting performance.

7. Concluding Remarks

This paper discussed our work on a framework named *Pin++*, which is used to write analysis tools for the dynamic binary instrumentation (DBI) tool named *Pin*. Our experience and results show that Pin++ improves several design metrics, such as cyclomatic complexity and modularity, while reducing instrumentation overhead. Because we have a framework like Pin++ that increases the level of abstraction for designing and implementing Pintools, we believe we can start answering *harder* questions with DBI tools such as those dealing with large volumes of instrumentation data, meaningful sampling, and multi-language instrumentation. Lastly, Pin++ has many other features not discussed in this paper due to space limitations, such as STL-like iterator for objects; ScopeGuards [26] to ensure paired functions are ordered correctly; and objects to represent exceptions and replacement functions in Pin[1].

Pin++ is freely available in open-source format at the following location: `github.com/SEDS/PinPP`.

References

[1] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *SIGPLAN Notes*, vol. 40, pp. 190–200, June 2005.

[2] K. Hazelwood, "Dynamic binary modification: Tools, techniques, and applications," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 2, pp. 1–81, 2011.

[3] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 2–11.

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

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

[6] M. Attariyan, M. Chow, and J. Flinn, "X-ray: automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 307–320.

[1] Details about these feature can be found in the Pin++ Github repository, and on its wiki (`github.com/SEDS/PinPP/wiki`).

[7] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 2003, pp. 265–275.

[8] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 15–28.

[10] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[11] Intel, "Pin Manual," http://software.intel.com/sites/landingpage/pintool/docs/61206/Pin/html.

[12] R. C. Martin, "Design Principles and Design Patterns," *Object Mentor*, pp. 1–34, 2000.

[13] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, December 1976.

[14] T. Hemmann, "Reuse Approaches in Software Engineering and Knowledge Engineering: A Comparison," in *Position Paper Collection of the 2nd Int. Workshop on Software Reusability*, no. 93-69, 1993.

[15] K. Czarnecki and U. W. Eisenecker, "Components and Generative Programming," in *Software Engineering–ESEC/FSE '99*. Springer, 1999, pp. 2–19.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[17] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[18] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: Concepts, tools, and techniques from boost and beyond*. Addison-Wesley Professional, 2004.

[19] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[20] W. H. Brown, R. C. Malveau, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York, NY: Wiley, 1998.

[21] J. O. Coplien, "Curiously recurring template patterns," *C++ Report*, vol. 7, no. 2, pp. 24–27, 1995.

[22] R. E. Schantz and D. C. Schmidt, "Middleware for distributed systems - evolving the common structure for network-centric applications," 2001.

[23] J. de Guzman, "Boost.Spirit," http://boost-spirit.com.

[24] ISO/IEC, "Summary of the ISO EBNF Notation," www.dataip.co.uk/Reference/EBNF.php.

[25] S. D. Huston, J. C. E. Johnson, and U. Syyid, *The ACE Programmer's Guide*. Boston: Addison-Wesley, 2002.

[26] WikiBooks, "More C++ Idioms," http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Scope_Guard.