Towards Real-time Monitoring of DRE Systems Using Dynamic Binary Instrumentation Middleware

Tanumoy Pati and James H. Hill
Dept. of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: {tpati, hillj}@cs.iupui.edu

***Abstract*—Dynamic binary instrumentatation (DBI) frameworks allow application developers and testers to non-intrusively collect trace profiles from their applications in production environments. The collected trace profiles are then used to analyze system behavior. Unfortunately, applying a DBI frameworks to an enterprise distributed real-time and embedded (DRE) systems is not a *trivial* process. This is because it is the end-user's responsibility to manually collect traces from each host, integrate the many (disconnect) execution traces, and provide meaningful analysis. Moreover, it is *hard* for DRE system developers and testers to dynamically control DBI frameworks to ensure minimal impact on quality-of-service (QoS) while collecting enough metrics to support analysis—especially for long running enterprise DRE systems.**

This work-in-progress paper therefore provides two contributions to instrumentation of enterprise DRE systems. First, this paper discusses how DBI frameworks can be integrated with dynamic instrumentation middleware to bridge the gap of autonomously collecting trace profiles and metrics from enterprise DRE systems in a non-intrusive manner. Secondly, this paper discusses what research questions we plan to address as a result of integrating DBI frameworks with dynamic instrumentation middleware.

***Keywords*-dynamic binary instrumentation, dynamic instrumentation middleware, enterprise DRE systems, Pin, OASIS**

I. Introduction

Validating quality-of-service (QoS) properties (*e.g.*, response time, latency, accuracy, throughput, synchronization and scalability) of an enterprise distributed real-time and embedded (DRE) system in its execution environment entails monitoring its behavior and resources, such as CPU utilization, memory usage, event arrival rate, and application heartbeat in real-time [10]. These metrics of interest are typically captured in trace profiles that can be used to track function calls (*e.g.* system or application- level calls) and examine/change arguments, examine/monitor resource usage, track application threads, and locate bugs.

The trace profiles can be generated using either *instrusive* or *non-instrusive* instrumentation techniques. Intrusive instrumentation is when the original application's source code is modified to collect the necessary metrics [11]. Whereas, non-intrusive instrumentation is when the original applications source code is not modified to collect the necessary metrics (*e.g.*, the binary code is transparently modified) [9]. This is a major benefit of using non-instrusive instrumentation on a software application.

One method that supports non-instrusive instrumentation of software systems, such as enterprise DRE systems, is *dynamic binary instrumentation (DBI)*. DBI is a general-purpose technique used to analyze the runtime behavior of a software/system, thereby facilitating dynamic program analysis tasks, *e.g.*, profiling, performance evaluation, optimization, and bug detection [13] [7]. Examples of DBI frameworks include DynamoRIO [2], Valgrind [8], Pin [6], and Dtrace [3].

Although DBI frameworks enable lightweight and non-instrusive instrumentation of an enterprise DRE system, one of their major shortcomings is that metrics collected by such frameworks are typically stored on local disk. In an enterprise DRE system, it is the responsibility of an end-user (*e.g.*, developer and tester) to either (1) manually collect the execution traces from each host in the distributed environment or (2) implement a proprietary framework to collect execution traces. The former approach is tedious, time-consuming, and error prone. The latter approach is a viable choice but as the framework uses system resources (such as CPU cycles), the overall performance of the system/application is significantly altered.

To overcome the shortcomings mentioned above, dynamic instrumentation middleware [4] can be used to collect metrics in a distributed environment. This work-in-progress paper therefore presents our current efforts and approach on realizing a solution where DBI frameworks are integrated with dynamic binary instrumentation middleware, such as the *Open-source Architecture for Software Instrumentation of Systems (OASIS)* [4]. We also discuss research questions we plan on addressing after integrating DBI frameworks with dynamic instrumentation middleware. Our believe is that this integration will enable real-time monitoring of enterprise DRE systems will enabling autonomous control to minimize impact of system QoS—especially for long running systems.

Paper organization. The remainder of this paper is organized as follows: Section II provides an overview of Pin; Section III presents a brief overview of OASIS and discusses the integration approach of Pintools and OASIS; and Section IV provides the concluding remarks.

II. Overview of Pin

Pin is a DBI framework for Linux and Microsoft Window's software applications. Pin operates by dynamically inserting code into arbitrary places in a software application under instrumentation to collect run-time information. It is an easy-to-use, portable, transparent, and efficient instrumentation platform [6].

Pin provides a rich application programming interface (API) that enables end-users to create instrumentation tools called *Pintools* in C/C++. The Pintools are basically tools are domain-specific plugins that accomplish different program analysis tasks (*e.g.*, profiling, performance evaluation and bug detection). Pin follows the model of *Analysis Tools using OM [12] (ATOM)*, which allows the user to easily instrument and analyze an executable at the instruction level without any prior knowledge of the underlying instruction set [5]. The *architecture independent* API of Pin therefore gives it a high degree of portability and allows Pintools to be source compatible across all supported instruction sets (*e.g.*, IA-32, Intel64, IA-64, ARM) and operating systems (*e.g.*, Linux, Windows, MacOS, FreeBSD) [14].

The just-in-time (JIT) compiler is utilized by Pin to insert and optimize code, while instrumentation techniques such as register reallocation, inlining, liveness analysis, and instruction scheduling for optimizing jitted code, help Pin to outperform other instrumentation tools, such as Valgrind and DynamoRIO [6]. Pin's efficiency can further be attributed to its *process attaching and detaching* technique wherein it first attaches itself to a process, instruments it and then detaches from it. This therefore makes Pin suitable for large, long-running applications because the instrumentation overhead is considerably low.

Conceptually, a Pintool consists of two components: *instrumentation routine* and *analysis routine*. Instrumentation routine is the portion of the Pintool that inspects the application's instructions and inserts calls to analysis routines. Analysis routines is the portion of the Pintool used when the program executes an instrumented instruction and gathers data (*i.e.*, defines the specific actions to perform after the instrumentation is activated) [1].

```
1
2  static UINT64 icount = 0;
3
4  // ANALYSIS ROUTINE
5  VOID docount() { icount++; }
6
7  // INSTRUMENTATION ROUTINE
8  VOID Instruction(INS ins, VOID *v) {
9    INS_InsertCall(ins,
10                  IPOINT_BEFORE,
11                  (AFUNPTR)docount,
12                  IARG_END);
13 }
14
15 // ...
16
17 // Called at application exit; writes results
18 // to a local file on disk.
19 VOID Fini(INT32 code, VOID *v) {
20  ofstream OutFile;
21  OutFile.open(KnobOutputFile.Value().c_str());
22  OutFile.setf(ios::showbase);
23  OutFile << "Count " << icount << endl;
24  OutFile.close();
25 }
26
27 int main(int argc, char * argv []) {
28  // Initialize pin
29  if (PIN_Init(argc, argv))
30    return -1;
31
32  // Register Instruction to be called to
33  // instrument instructions
34  INS_AddInstrumentFunction(Instruction, 0);
35
36  // Register Fini to be called when the
37  // application exits
38  PIN_AddFiniFunction(Fini, 0);
39
40  // Start the program, never returns
41  PIN\_StartProgram();
42
43  return 0;
44 }
```

Listing 1. An example Pintool for counting instructions written in C++.

Listing 1 illustrates an example Pintool for counting the instructions of an application. As shown in this example, the instrumentation routine `Instruction()` is called every time a new instruction is encountered. Likewise, the analysis routine `docount()` is called each time the `Instruction()` method is invoked. When the application exits, the `Fini()` method is invoked. At this point, the Pintool writes the results to an output file on local disk.

Finally, Pin, the Pintool and, the application all share the same address space while executing. These three entities are explicitly invoked by the user along with their corresponding run-time arguments by the following syntax on command line:

```
%> pin [Pin Args] [-t <Pintool> [Pintool Args]] \
     <App> [App args]
```

The remainder of this work-in-progress paper will discusses our approach for integrating Pin with the dynamic instrumentation middleware named OASIS to support real-time monitoring of enterprise DRE systems.

III. Integrating Pin with OASIS

This section discusses our approach for integrating Pin with OASIS to provide real-time instrumentation and monitoring capabilities for enterprise DRE systems.

A. Brief Overview of OASIS

OASIS is a dynamic instrumentation middleware for DRE systems that facilitates the ability to handle (*e.g.* collect, extract, and analyze) metrics without a priori knowledge of metric details [4]. Figure 1 presents a high-level overview diagram of OASIS's architecture. As shown in the figure, OASIS consists of five major components:

- **Software probe**, which are autonomous agents that collect both system and application-level metrics, such as the state of an application, number of events it sends/receives, and current memory usage.
- **Embedded Instrumentation Node (EINode)**, which is deployed one per application-context, *i.e.*, a domain of commonly related data, and receive metrics from the software probes.

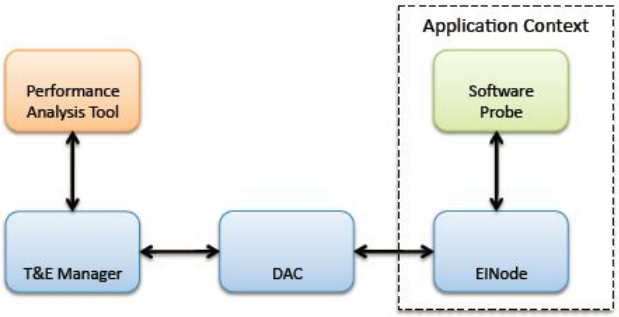

Fig. 1. High-level overview of OASIS's architecture [4].

- **Data Acquisition and Controller (DAC)**, which is a persistent database with a fixed location in the target environment that can be located via a naming service. It receives data from the EINode and archives it for later acquisition by the performance analysis tools.
- **Test and Evaluation (T&E) Manager**, which acts as the entry point for user applications into OASIS. It accumulates data from the various DAC's that have registered with it, along with enabling user applications to alter the runtime behavior of probes by sending signals to it.
- **Performance Analysis Tools**, which are domain-specific tools that interact with OASIS by requesting metrics collected from different software probes via the T&E manager for analyzing the system/application performance. These tools hold the ability to alter the runtime behavior of probes via signals, thereby allowing system developers to control the effects of software instrumentation at runtime and also minimizing their impact on the overall system performance.

B. Approach for Integration

Each software probe has a Pintool associated to it, which performs various tasks such as collecting instruction and basic blocks count, profiling, cache simulation, trace analysis, and memory bug checkers. Inside the Pintool code, the software probe is first initialized, and then it is registered with a respective EINode. Once this EINode is activated, the software probe is forced to flush out the most recently collected data to its hosting EINode, which is ultimately stored in the DAC.

As shown in the Figure 2, we can instantiate multiple software probes inside the Pintool that sends data to an EINode. For an enterprise DRE system, the data collected by the dynamic instrumentation tool (i.e., Pintool) is no longer collected on the local host's disk; instead, it is sent to the DAC with a static location in real-time.

```
1  static size_t icount = 0;
2
3  // OASIS constructs
4  static Pin::Instruction_Counter_Probe_Impl count_probe;
5  static OASIS::Standard_EINode einode;
6
7  // ANALYSIS ROUTINE
8  VOID docount() { count_probe.count (++ icount); }
9
10 // INSTRUCTION ROUTINE
11 VOID Instruction(INS ins, VOID *v) {
12   INS_InsertCall(ins,
13                  IPOINT_BEFORE,
14                  (AFUNPTR)docount,
15                  IARG_END);
16 }
17
18 // This function is called when the application exits
19 VOID Fini(int code, VOID *v) {
20   // Flush probe data to DAC.
21   count_probe.flush();
22   einode.deactivate();
23   einode.destroy();
24 }
25
26 int main (int argc, char * argv[]) {
27   // Initialize pin
28   if (PIN\_Init (argc, argv))
29     return -1;
30
31   INS_AddInstrumentFunction(Instruction, 0);
32
33   // Register Fini to be called when the application exits
34   PIN_AddFiniFunction(Fini, 0);
35
36   // Initialize OASIS constructs
37   einode.init(argc, argv);
38   einode.register_probe("Counter", &count_probe);
39
40   // Activating the EINode
41   einode.activate();
42
43   // Start the program, never returns
44   PIN\_StartProgram();
45
46   return 0;
47 }
```

Listing 2. The instruction counter Pintool that has been integrated with OASIS.

Likewise, Listing 2 shows the instruction counter Pintool from Listing 1 that has been integrated with OASIS. As shown in this listing, a probe called `Instruction_Counter` has been integrated into the Pintool that collects the instruction count for a software application. The probe first registers itself with the EINode and then it is activated. Whenever a new instruction is encountered, the instrumentation routine named `Instruction()` is invoked. This, in turn, calls the `docount()` analysis routine, which updates the software probe's state. Finally, when the application is exiting, the software probe is forced to flush out the data to the respective EINode.

Unlike a traditional Pintool, when a Pintool is integrated with OASIS the analysis routines do not perform any "real" analysis. Instead, the data used in the analysis routine is collected and stored in the DAC. Instead, OASIS's performance analysis tools are responsible for performing the analysis that was once done by the Pintool. This therefore help decouple the instrumentation and analysis aspect of a DBI framework. Finally, because the software probes can be controlled remotely, it is possible to dynamically turn on/off different instrumentation points that have been tied to a software probe in real-time.

IV. Concluding Remarks

This work-in-progress paper presented our approach for integrating DBI frameworks, such as Pin, with dynamic instrumentation middleware, such as OASIS. This will enable DRE systems and testers to non-intrusively instrument and monitor

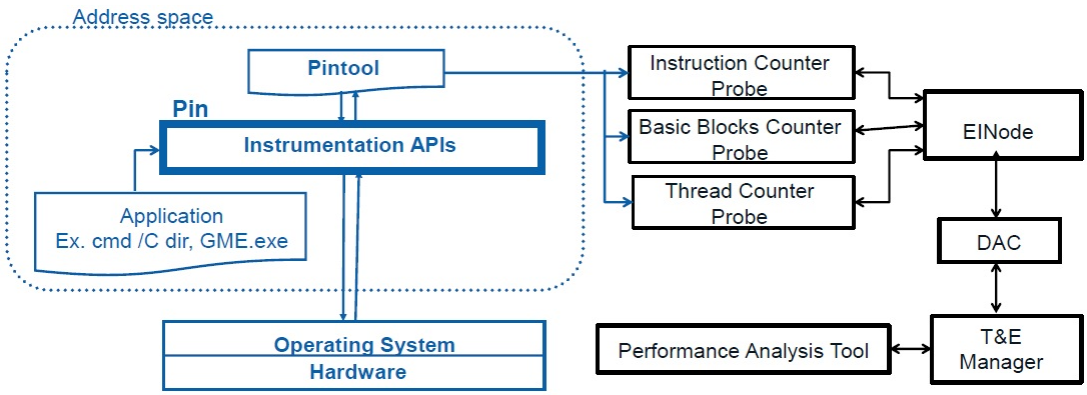

Fig. 2. A Pintool integrated with three software probes that communicate with the OASIS middleware.

enterprise DRE systems in real-time. Based on this integration, we will be able to investigate research questions related to patterns of software instrumentation and real-time and adaptive monitoring while minimizing impact on overall QoS. We will investigate these research questions in the context of real-time enterprise DRE systems from the domain of shipboard computing.

REFERENCES

[1] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, 2010.

[2] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[3] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *6th Symposium on Operating Systems Design and Implementation 2004*, 2004.

[4] J. H. Hill, H. Sutherland, P. Staudinger, T. Silveria, D. C. Schmidt, J. M. Slaby, and N. Visnevski. OASIS: An Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems. *International Journal of Computer Systems Science and Engineering, Special Issue: Real-time Systems*, April 2011.

[5] Intel. Pin manual. http://www.pintool.org/docs/41150/Pin/html/.

[6] 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*, 40:190–200, June 2005.

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

[8] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *SIGPLAN Not.*, pages 89–100, June 2007.

[9] O.Cole. Aprobe: A non-intrusive framework for software instrumentation, 2004.

[10] D. C. Schmidt, R. Schantz, A. Gokhale, and J. Loyall. Middleware r&d challenges for distributed real-time and embedded systems. In *ACM SIGBED Rev.*, pages 6–12, April 2004.

[11] B. Sridharan, B. Dasarathy, and A. P. Mathur. On building non-intrusive performance instrumentation blocks for corba-based distributed systems. In *Proc. IEEE Int. Computer Performance and Dependability Symp. IPDS 2000*, pages 139–143, 2000.

[12] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

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

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