Dynamic Binary Instrumentation Framework for CE Devices

Alexey Gerenkov
SRC Moscow, Samsung Electronics
`a.gerenkov@samsung.com`

Sergey Grekhov
SRC Moscow, Samsung Electronics
`grekhov.s@samsung.com`

Jaehoon Jeong
SAIT, Samsung Electronics
`hoony_jeong@samsung.com`

Abstract

Developers use various methods and approaches to find bugs and performance bottlenecks in their programs. One of the effective and widely used approach is application profiling by dynamic instrumentation. There are many various tools based on dynamic instrumentation. Each tool has its own benefits and limitations what often forces developers to use several of them for profiling. For example, in order to use `Kprobe`-based [1] Systemtap [2] tool developers need to write instrumentation script using special language. To use `Dyninst` [3] profiling library developers need to write instrumenting programs in C++. Thus each tool realizes its own profiling technology. Additionally various profiling tools produce output data in their own formats and those formats are incompatible. Thus two above problems significantly increase complexity of debugging.

In this paper we describe unique dynamic binary instrumentation engine concept which is used in our monitoring tool — System-Wide Analyzer of Performance (SWAP). This tool has modular open architecture and API which allow integrating various tools for providing powerful instrumentation and analysis framework for developers. `Dyninst` and `Kprobe`-based instrumentation engines are integrated into SWAP framework and used in a similar way. Modular structure of SWAP can be extended with other instrumentation and analysis methods by easy way. Also SWAP has several levels of API: instrumentation API, connection API, control API, user interface API and monitoring language framework API. This multilevel API architecture allows developers to re-use SWAP functionality and embed it into their own solutions. All above mentioned SWAP advantages essentially simplify debugging profiling process for embedded software.

1 Introduction

During the last decade computer market has been shifting its focus from PCs to consumer electronic devices that have made great steps in increasing their functionality. Now CE embedded systems can provide wide set of capabilities for user. Devices interact directly with consumers and quality of their work influence manufacturer's brand greatly. So bugs and ineffectiveness of software are very critical for CE products. It is agreeable that Linux has become popular as a platform for modern embedded systems. However, it still has issues to be solved due to limited resources (i.e. CPU power & memory size) and absence of network interface in the embedded environment. Linux developers use various methods and approaches to find bugs and performance bottlenecks in their programs. One of the effective and widely used approach is application profiling by dynamic instrumentation.

There are a number of tools based on dynamic instrumentation for *nix systems. Every tool has its own list of supported features that can be split into the following categories:

- Supported operating systems, kernel versions;
- Supported processor architectures;
- Instrumentation capability (functions entries/exits, functions bodies, certain types of instructions etc.);
- Type of collected information (variables, stack, user or kernel space data etc.);
- Instrumentation overhead;
- Format of collected profiling data;

Set of features supported by particular tool is limited, and the list of features provided by tool determines its advantages and disadvantages for developers. For example, well-known `Kprobe`-based tool `Systemtap` which provides powerful script language for dynamic instrumentation does not support MIPS architecture. Another well-known tool `LTTng` [4] supports great set of architectures (including MIPS), but its instrumentation capabilities are limited by hooks inserted into source code.

Since every tool has limited set of supported features developers should often make choice what tool to use in certain circumstances. The problem of choosing a suitable tool for dynamic instrumentation and profiling is widely discussed over the technical forums in the internet, Recently a comparison report [5] has been published for the tools mentioned above and `DTrace` [6], another instrumentation tool for Solaris, Mac OS X, BSD and QNX. All those discussions and comparison report were intended to provide developers with summarized view of available profiling functionality on different platforms.

As it was mentioned above features provided by single tool can be insufficient for developers, so it is reasonable to consolidate the advantages of multiple tools simultaneously to perform one experiment: typical case which usually requires the using of several tools is instrumenting both user and kernel space. But using several tools on CE devices is not quite convenient due to the following reasons:

- Some tools can not be executed on CE devices in standalone mode, due to device resource limitations, so experiments which need to run multiple tools simultaneously can not be performed;
- Not all tools support host-target architecture which is preferable for resource limited CE devices, so experiments which need to run multiple tools simultaneously can not be performed;
- Using multiple tool complicates instrumentation process, because instrumentation scripts/programs needs to be written for several tools;
- All tools have different format of collected data, so additional scripts/programs are needed to bring different output data to one format which can be used for analysis;
- Quite often data, collected using several tools, should be merged and synchronized in order to create ordered sequence of events which reflects the essence of experiment;

All above problems significantly increase complexity of profiling using multiple tools.

Let's consider the following example (see Figure 1) of profiling programs using multiple tools. Developer wants to analyze how many page faults are generated by memory accesses made by considered program and if it is possible to find access patterns which can be optimized. For this experiment developer needs to profile all memory access made by his program and all calls to `do_page_fault` kernel function. To instrument memory accesses he uses well-known tool — `Dyninst`, to instrument `do_page_fault` `Systemtap` is used. In order to use `Dyninst` engineer has to write special instrumentation program in C++, compile it and link with `Dyninst` library. For using `Systemtap` developer needs to write instrumentation script using special language. After gathering necessary data by both tools one needs to take care about merging data sets after experiment. This is not enough convenient for developer because of to perform data analysis developer needs to create a analyzing script which needs to parse data and represent it as objects for further processing. Finally, all these actions can be available if and only if CE device has enough resources for instrumenting the considered application in standalone mode. Otherwise, one should take care about remote instrumentation using typical host-target model.

Thus, using Universal Instrumentation Engine could significantly simplify using multiple dynamic instrumentation tools (see Figure 2):

- Instead of working with several separate tools developer uses universal instrumentation API which helps to avoid writing handlers with different languages and perform several linking/compiling procedures;
- Universal Instrumentation API encapsulates unique instrumentation technique for each tool and provides a unified method of using each DBI engine;

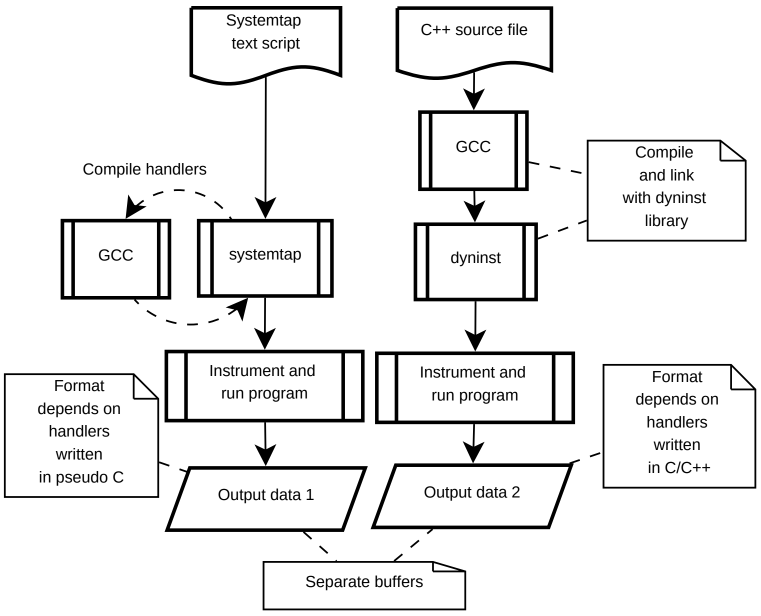

Systemtap text script
C++ source file
GCC
Compile and link with dyninst library
Compile handlers
GCC
systemtap
dyninst
Instrument and run program
Instrument and run program
Format depends on handlers written in pseudo C
Format depends on handlers written in C/C++
Output data 1
Output data 2
Separate buffers

Figure 1: Using Multiple Instrumentation Tools

- Universal engine provides requested dynamic instrumentation in standalone or remote mode using as less resources for multiple tools as possible;

- All handlers of multiple tools are written similarly in sense of output data format, so the collected trace of events will be, firstly, saved in unified data format and, secondly, provided in a merged state, thus avoiding developer from taking care of problems regarding parsing/merging different format data;

This approach eliminates disadvantages of process of using multiple instrumentation engines described above. Universal engine integrates several instrumentation methods in order to provide developer with ability to run his instrumentation scenarios using several methods through one universal interface. It also provides gathered data in unified format what simplifies post-mortem data analysis.

2 Framework Description

In this paper we describe unique dynamic binary instrumentation engine concept which is used in our monitoring tool — System-Wide Analyzer of Performance (SWAP). This tool was designed for profiling of user and kernel space software on embedded Linux systems. It provides extensible framework for system profiling and analysis of gathered data. It uses dynamic instrumentation technique for collecting data about system behavior. SWAP integrates several instrumentation tools and provides developers with ability to use multiple tools for profiling applications on CE devices. It has open modular design which allows other tools (e.g. front-ends, GUIs) to re-use its functionality. To fit the needs of various embedded platforms SWAP also supports host-target communication concept and standalone mode of operation when there is no connection to host. Finally, it provides a simple and convenient toolkit for post-mortem data analysis based on Python scripts which can be easily extended on the needs of user.

2.1 Architecture

In general profiling of applications consists of the following steps:

1. Developer specifies probing points and handlers which should be attached to those points;

2. Instrumentation engine resolves probing points in application binary and prepares instrumentation code to be inserted into application;

3. Instrumentation code is transferred to target in case of host-target architecture;

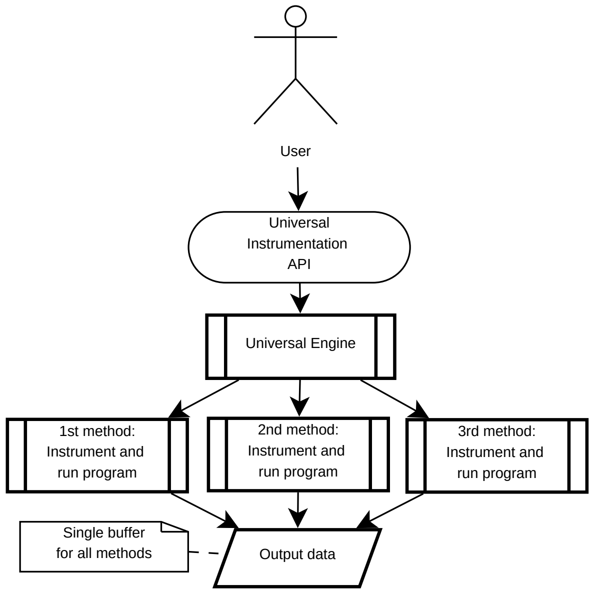

User
Universal Instrumentation API
Universal Engine
1st method: Instrument and run program
2nd method: Instrument and run program
3rd method: Instrument and run program
Single buffer for all methods
Output data

Figure 2: Universal Instrumentation Engine

4. Engine inserts instrumentation code into application and runs it if necessary;

5. Profiling data are collected by instrumentation handlers;

6. Engine removes instrumentation code;

7. Collected data is transferred from target to host in case of host-target architecture;

8. Collected data are stored in intermediate storage;

9. Developer analyzes collected data;

SWAP has open modular architecture (see Figure 3) which allows developers to re-use it for building their own tools. SWAP instrumentation framework consists of the following main components which implement above profiling scenario for embedded devices and provide the full set of capabilities for dynamic instrumentation and further data analysis:

- ***Instrumentation Engine.*** This is essential component of SWAP instrumentation framework which fully implements the functionality of described above Universal Instrumentation Engine. It provides API for instrumentation of user and kernel space functions using various instrumentation methods. It allows to instrument entry/exit and body of specified functions. Also it provides API to configure and control profile data collection. It consists of two parts: host and target. Host Instrumentation Engine performs step 2 from above scenario. It prepares data necessary for instrumentation, for example, resolves function addresses and prepares instrumentation code to be inserted. Target Instrumentation Engine performs steps 3 and 4 from above scenario. It does essential work: inserts instrumentation into application/kernel and collects data. Currently engine integrates two instrumentation methods: `Kprobe`-based and `Dyninst`-based. For kernel instrumentation `Kprobe`-based method is used. For user space `Kprobe` or `Dyninst` methods are used depending on what program points should be instrumented (e.g. functions or memory access instructions).

- ***Communication Agent.*** This component provides API for connecting to target and transferring commands and responses to target and obtaining collected data from target. It consist of two parts: host and target. The component encapsulates SWAP target communication protocol. This component is used by instrumentation engines to control instrumentation process and data transfer. In standalone mode this component is not used.

- ***Collected Data Storage.*** This is storage for collected data. It stores collected data and provides effective methods for retrieval of collected information when data are analyzed. SWAP supports several modes of data collection: normal mode when all data are collected on target and after experiment they are transferred to host and continuous mode when data are transferred to host continuously as they appear. These two modes allows to user to choose optimal data collection policy for his experiment. Normal mode has less overhead for target, but it is limited by size of buffer on target for collected data. So in normal mode amount of profiled data are limited by size of buffer on target. Continuous transfer mode has no such limitation. Buffer is used as intermediate storage before data are transferred to host. But this mode has greater overhead because of data transmission.

- ***Data Analyzing Framework.*** As a final step of resolving considered problem, SWAP provides to developers analyzing framework which allows easy data manipulating and analyzing. This component provides framework with API for loading, parsing and analyzing data kept in Collected Data Storage. Developers can easily extend this framework and reuse its components in their own analyzing scripts written in SWAP Python-based language.

The described components cover all basic functionality needed for monitoring with help of dynamic instrumentation: ability to instrument kernel space and user space, ability to use standalone mode or host-target model, ability to save collected data in unified format and, finally, perform data analysis for understanding considered behavior of the CE device. Let's consider general SWAP profiling scenario in details (see Figure 3):

- Upon user actions SWAP GUI configures instrumentation and starts it via instrumentation API provided by engine. Instrumentation configuration includes functions to be instrumented, size of buffer for collected data etc;

- Host instrumentation engine prepares information necessary for profiling, connects to target via host communication agent and sends data to target;

- Instrumentation agent on target inserts instrumentation code into application and starts it if necessary;

- Data collection started;

- When user stops data collection (via SWAP GUI) collected data are transferred back to host via communication agents and saved in data storage;

- Then when user wants to visualize results of profiling SWAP GUI uses analyzing scripts based on Data Analyzing Framework to calculate and represent various statistics;

- SWAP visualizes results;

2.2 Instrumentation API

As it was mentioned above SWAP instrumentation engine provides to developers universal instrumentation API. This API provides common way for profiling programs using different instrumentation methods. As it was said before currently there are two methods are supported by SWAP: `Kprobe`-based and `Dyninst`-based. This API provides general set of functions which does not depend on underlying instrumentation engine. API allows to:

- Add/remove probe for specified kernel function. Kernel function probes trace entry and exit from functions;

- Add/remove probe for specified user space application's function. User space function probes trace entry and exit from functions;

- Add/remove probe inside functions. Function body probes trace execution of instruction at specified address inside functions;

- Add/remove memory access probe inside functions. Memory access probes trace execution of memory access instructions inside functions;

- Configure target buffer size;

- Set filters by process IDs and names;

- Define conditions for data collection start and end;

- Start and stop profiling process;

- Retrieve list of libraries used by application;

- Retrieve list of application/library functions;

- Retrieve list of kernel functions;

- Configure data transfer mode for host-target model (one-time or continuous);

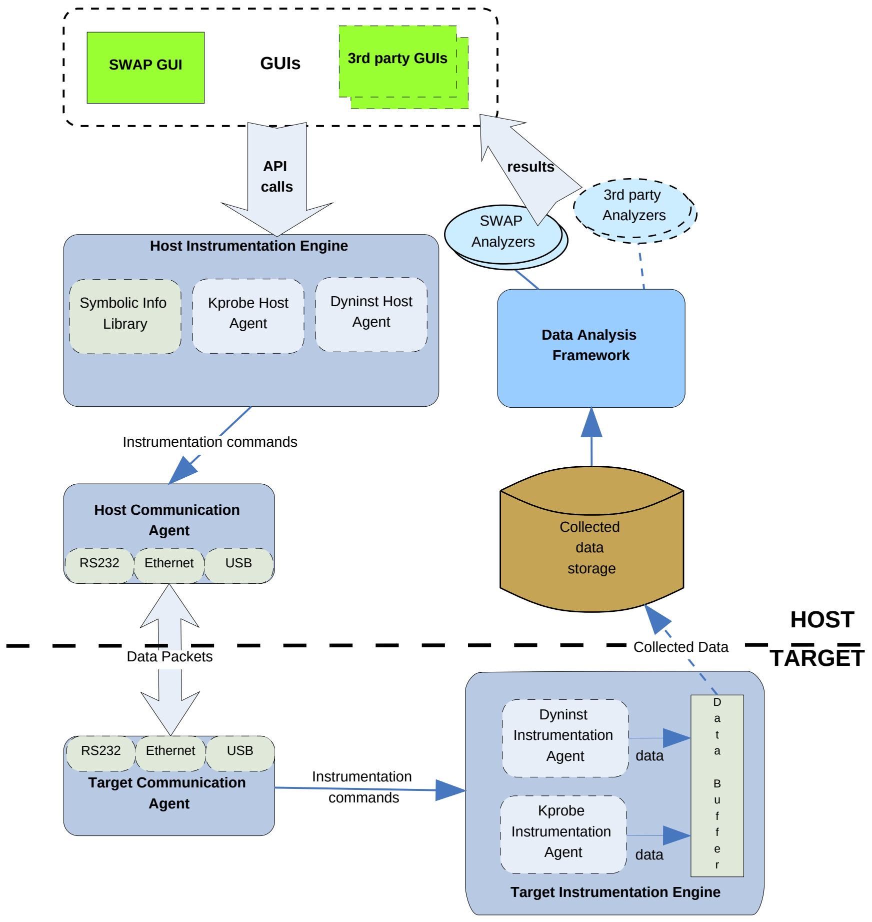


Figure 3: SWAP Architecture

3 Profiling With SWAP

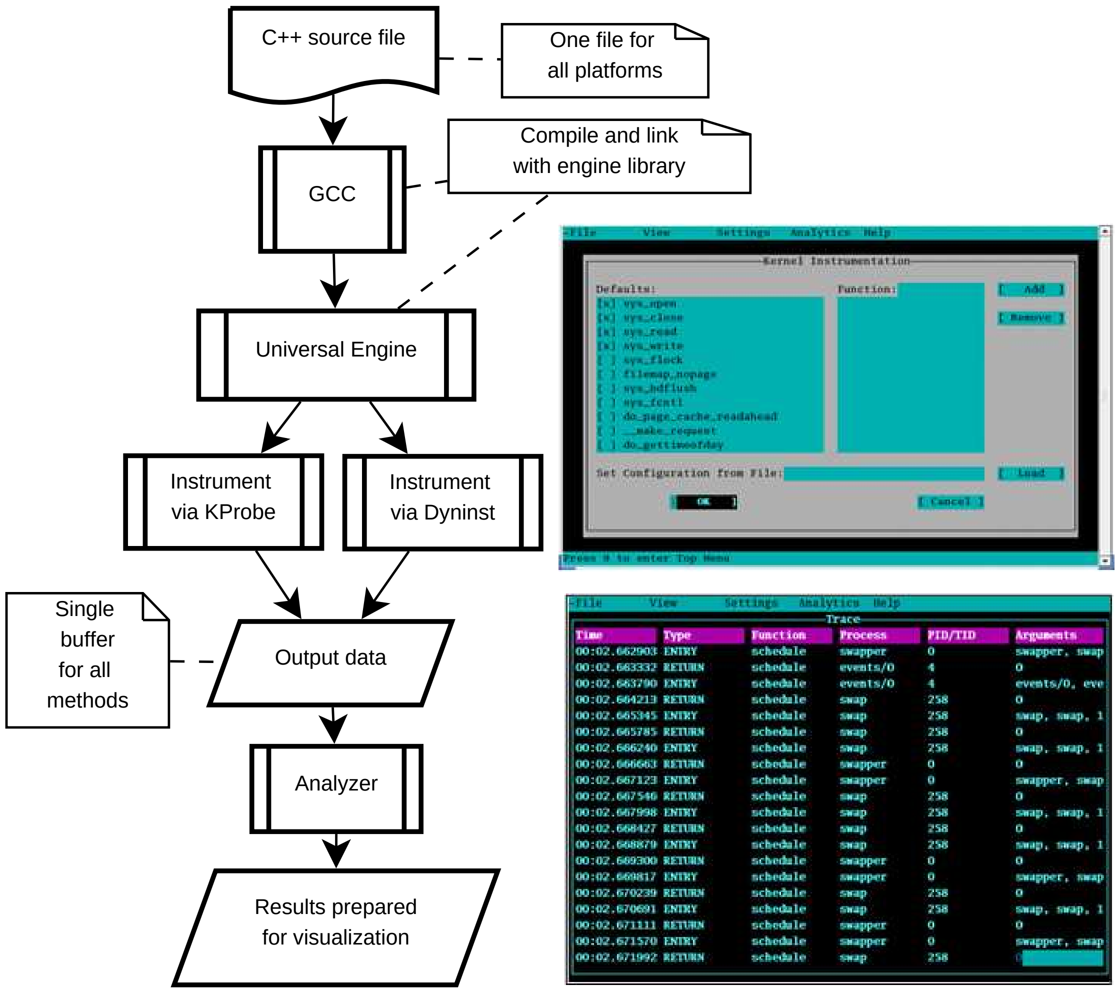
There are several ways in which developers can use SWAP instrumentation framework for profiling:

- They can use it as library in order to build their own tools on top of the engine;
- Developers can use SWAP GUI front-end itself for instrumentation, control and visualization of results;

As it was described earlier SWAP instrumentation engine is universal instrumentation engine which integrates several instrumentation tools (see Figure 4). SWAP GUI follows general approach of building tools on top of SWAP instrumentation engine. It uses it as shared library. Also it extends SWAP data analyzing framework with a set of own scripts which analyze collected data and produce data for visualization of various statistics. Let's see how SWAP solves problems raised by usage of multiple tools. We will consider example described in introduction where memory accesses and `do_page_fault` kernel function should be instrumented for the same application behavior (developer wants to analyze how many page faults are generated by memory accesses made by considered program and if it is possible to find access patterns which can be optimized).

Steps which should be performed to instrument program using multiple tools `Systemtap` and `Dyninst`:

1. Write `Systemtap` script for kernel functions instrumentation using a built-in language;
2. Compile written handlers for kernel functions;
3. Instrument kernel functions and run considered program;
4. Collect and save essential data;



```
Kernel Instrumentation
Defaults:                                 Function:
[x] sys_open
[x] sys_clone
[x] sys_read
[x] sys_write
[ ] sys_flock
[ ] filemap_nopage
[ ] sys_bdflush
[ ] sys_fcntl
[ ] do_page_cache_readahead
[ ] __make_request
[ ] do_gettimeofday
Set Configuration from File:
Press # to enter Top Menu
```

```
Trace
Time            Type      Function   Process    PID/TID   Arguments
00:02.662903 ENTRY     schedule   swapper    0         swapper, swap
00:02.663332 RETURN    schedule   events/0   4         0
00:02.663790 ENTRY     schedule   events/0   4         events/0, eve
00:02.664213 RETURN    schedule   swap       258       0
00:02.665345 ENTRY     schedule   swap       258       swap, swap, 1
00:02.665785 RETURN    schedule   swap       258       0
00:02.666240 ENTRY     schedule   swap       258       swap, swap, 1
00:02.666663 RETURN    schedule   swapper    0         0
00:02.667123 ENTRY     schedule   swapper    0         swapper, swap
00:02.667546 RETURN    schedule   swap       258       0
00:02.667998 ENTRY     schedule   swap       258       swap, swap, 1
00:02.668427 RETURN    schedule   swap       258       0
00:02.668879 ENTRY     schedule   swap       258       swap, swap, 1
00:02.669300 RETURN    schedule   swapper    0         0
00:02.669817 ENTRY     schedule   swapper    0         swapper, swap
00:02.670239 RETURN    schedule   swap       258       0
00:02.670691 ENTRY     schedule   swap       258       swap, swap, 1
00:02.671111 RETURN    schedule   swapper    0         0
00:02.671570 ENTRY     schedule   swapper    0         swapper, swap
00:02.671992 RETURN    schedule   swap       258       0
```

Figure 4: SWAP Profiling Scheme

5. Write C++ program which uses `Dyninst` for instrumentation of memory accesses;
6. Compile and link considered program with `Dyninst` library;
7. Run instrumented program second time;
8. Collect and save essential data;
9. Merge data collected by both tools;
10. Write program/script for data analysis, including data parser, data analysis itself and formatted output information;

In comparison with previous procedure, developer should perform much less steps to instrument program using SWAP:

1. Write C++ program which uses SWAP engine for instrumentation of memory accesses and kernel functions using two or more DBI engines simultaneously;
2. Compile and link considered program with engine library;
3. Run program only one time;
4. Collect and save essential data using standalone mode or host-target mode;
5. Write program/script for data analysis itself, avoiding time consuming development of data parsing and formatting of output information;

As it can be easily seen, in this example, SWAP instrumentation procedure:

- is more simple, universal and resource saving due to unified approach for instrumentation needs less compilations/linking of instrumentation handlers;
- produces synchronized data which do not need merging due to simultaneous use of two or more DBI engines;
- provides more correct conditions of experiment due to one-time launch of considered program;
- allows perform considered experiment under host-target mode or standalone mode, thus extending abilities under strict hardware conditions (lack of memory for saving data or unavailable network connection);
- avoids developer of implementing data parsing during data analysis, thus giving ability to concentrate on analysis itself;

In more complex instrumentation scenarios developer may spend significant time for writing `Systemtap` script and `Dyninst`-based tool.

4 Conclusion

There are a number of instrumentation tools which have their advantages and disadvantages. There is no ideal tool which will satisfy all developer's needs. So developers have to duplicate and maintain their instrumentation scenarios in various forms which are supported by every tool.

In this paper we presented our tool SWAP which is built on top of universal profiling framework which can be extended by users analyzing modules. SWAP framework makes use of universal instrumentation engine concept in order to integrate functionality of several instrumentation tools and provide developers with universal and effective way of profiling programs on various embedded platforms using different instrumentation methods and analyzing collected profiling data.

References

[1] William E. Cohen, Gaining insight into the Linux kernel with Kprobes, RedHat Magazine, Issue#5, March 2005

[2] Frank Ch. Eigler. Problem solving with systemtap. In Proceedings of the Ottawa Linux Symposium 2006, 2006.

[3] Giridhar Ravipati, Andrew R. Bernat, Nate Rosenblum, Barton P. Miller and Jeffrey K. Hollingsworth, Toward the Deconstruction of Dyninst, Technical Report, Computer Sciences Department, University of Wisconsin, Madison, 2007.

[4] Mathieu Desnoyers, and Michel R. Dagenais, The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux, Linux Symposium Proceedings, Volume 1, Ottawa, Ontario, 2006.

[5] Systemtap, Dtrace, LTTng Comparison. `http://sourceware.org/systemtap/wiki/SystemtapDtraceComparison`

[6] Richard McDougall, Jim Mauro, and Brendan Gregg, Solaris Performance and Tools: Dtrace and Mdb Techniques for Solaris 10 and Opensolaris, Prentice Hall, 2006.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada