Branch Bitstream

Machine Instruction-level System Tracing

Pipat Methavanitpong, Tsuyoshi Isshiki, Dongju Li and Hiroaki Kunieda
Department of Communications and Computer Engineering
Tokyo Institute of Technology
Tokyo, Japan
{ppmet.th, isshiki, dongju, kunieda}@vlsi.ce.titech.ac.jp

***Abstract*—Tracing is an approach used for debugging and profiling software. Conventional approach to create traces is to instrument program's code tapping data out from an execution. An original execution is mixed with tracing routines. In consequent, a program in production state and development state are not identical. In addition, instrumentation code adds execution overhead and increases error probability to a traced program. Branch Bitstream (BB) aims to solve aforementioned drawbacks by partitioning tracing routines to an execution environment instead. Traces are generated without modifying a program. A trace-driven simulator uses a trace file and a program to perfectly reconstruct previous execution. This work implemented this concept in QEMU 0.14 for TCT architecture. Linux 3.0.0-rc7 booting was used as a profiling example. Emulation overhead is 2 times longer than one without BB in the current implementation.**

***Index Terms*—Tracing, Debugging, Profiling, Embedded Systems, Trace-driven Simulation**

I. INTRODUCTION

Tracing is an approach used for debugging and profiling software. Traces or clues are left along with an execution of a program. Developers line up these traces and use deductive reasoning to understand what happened previously. A bug can be picked out by finding an undesired state in the execution chain. Visualizing traces into easily understandable formats can ease developers making decisions to improve their programs.

Traces are generated by modifying a running context to perform tracing routines. The routines are conventionally instrumented in software i.e. inserting tracing code. Listing 1 shows a simple C program with `printf()` generating traces about how many times this loop has been repeated. Software instrumentation can be done in various stages: source code, at compile time, post link time or during run-time [1].

```
int ret, count;
do {
    ret = useful_func();
    printf("Run for #%d times\n", ++count);
}while(ret == 0);
```

Listing 1. A simple trace example by using `printf()`

Tracing can save time by removing a need to rerun a program. In addition, it is hard to recreate sporadic events. Once it is traced, the event exists forever for inspection. It also promotes collaboration in team as traces can be distributed.

However, tracing also has drawbacks [2]. It requires instrumentation. Software instrumentation adds extra code. It means there are more works to do in a program. It slows a traced program down. Instrumentation can be a source of errors; for example, the count variable is not initialized in Listing 1. Produced traces can mislead developers. Traces are generated overtime. If traces are recorded in a file, the file grows rapidly. It seems not possible for long-running programs if there is no mean to extinguish previous traces from a storage. Moreover, larger traces mean it takes longer to time to process into useful information.

Since the nature of embedded computers is being built with customized components, available or affordable tools required to create software for an embedded product may not exist or not complete a production flow. It is likely that these tools are developed in-house. This can be an error source to a built program, because the tools are not rigorously tested. Tracing code, either added by hands or tools, can be a problem too. Moreover, embedded computers are designed with constraints including time constraints. Executing extra tracing code might fail the constraints as in Fig. 1.

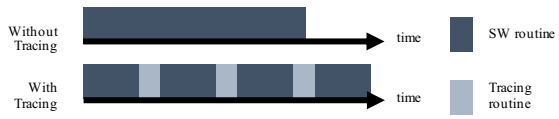

Fig. 1. Tracing routines intervene original behavior

Branch Bitstream (*BB*) aims to solve the uncertainties in software instrumentation by partitioning tracing routines to an execution environment instead. It preserves original software execution behavior. A program under development and a program in production phase becomes identical. It cuts the probe-effect of software instrumentation from error sources. If *BB* is implemented in hardware, a program can run with none or little overhead, because tracing routines are hardwired.

This paper is organized as following. Section II discusses about related works. Section III describes *Branch Bitstream* concept. Section IV shows how the *BB* concept is implemented in *QEMU*. Section V profiles Linux kernel booting by using *BB* traces from Section IV and the trace-driven simulator [3].

978-1-4799-8565-4/15/$31.00 ©2015 IEEE

Section VI discusses about the current state of implementation and future directions of possible improvement. And a conclusion of this work is given in Section VII.

II. Related Works

Tracing routines are usually implemented done in software, because the nature of software is more portable to other machines and flexible to changes than hardware's.

There are two types of software instrumentation:

1) Static instrumentation – tracing routines are statically bound to program code
2) Dynamic instrumentation – tracing routines are dynamically enabled and disabled

Several are open-source projects. This section introduces *Gcov*, *Ftrace*, *GDB* and *perf branch* briefly.

A. GNU Coverage

GNU Coverage (*Gcov*) is a static instrumentation type. It traces which lines were executed. Its routines are inserted by *GCC* with compiling option: `-fprofile-arcs` and `-ftest-coverage`. `-fprofile-arcs` places routines at branches and calls. `-ftest-coverage` places routines at every line of code. After executing an instrumented program, a trace file with an extension *gcda* is created. This trace file is used with its source code to visualize covered lines, branch taken percentage and the number of calls.

B. Function Trace

Function Trace (*Ftrace*) is a Linux kernel tracer instrumenting at every kernel function entries and exits. It is chosen to be included during kernel build configuration. It is originally a static type instrumentation, but it is now capable of dynamic instrumentation. Its dynamic instrumentation induces low overhead when it is disable. There is a simple condition check whether to perform tracing. If tracing is disabled, `nop` machine code is executed instead.

C. GNU Debugger

GNU Debugger (*GDB*) is a matured debugger for many programming languages. Despite of its name, it also has an ability to trace by injecting instrumentations dynamically. Tracing routines can be defined for each instrumentation. This is possible by taking an advantage of *ptrace* system call and signalling. A parent process, in this case is *GDB*, has a control over a child process, in this case is a target binary, both execution and states.

D. perf branch

perf branch is an experimental branch of a performance profiling tool, *perf* [4]. *perf branch* tracks how program flows by noting branch decisions. Branch decisions are recorded automatically by Intel *Branch Trace Store* (BTS) [5]. Intel *BTS* has been implemented from Intel Pentium 4 onwards. When it hits a branch, it records the branch's source address, the branch's destination address and whether the branch is predicted. Each of these is 32-bit long for 32-bit mode and 64-bit long for 64-bit mode, so the size of a record of these three is 12 bytes or 24 bytes, respectively. It is stored in a software designated buffer pointed by *Debug Store* (*DS*) *save area*. It makes an interrupt if *BTS interrupt threshold* is reached. This threshold must be large enough to allow pending interrupt to be handled before the buffer is full. *perf branch* collects *BTS* records and *memory map* (*mmap*) information. It uses *mmap* to resolve symbols where a branch jumps to, so it can visualize where a branch went in virtual address and destination file forms.

III. Branch Bitstream

A. Collecting Traces

Branch Bitstream (*BB*) tries to achieve tracing in Fig. 2 manner. It migrates tracing routines usually written in software as in related works in Section II to an execution environment.

Fig. 2. Tracing routines are migrated to execution environment

To perform this, *BB* must understand what happens in a traced program. Software may have many abstraction levels, but in the end they are translated into machine code. Machine code is the only interface that makes computers work. *BB* inspects a program execution at machine-instruction level.

It is tedious to record all executed instructions, and a trace file will become enormously large. Most machine instructions have linear execution flow. Only few changes it. *BB* records only dynamic control flow instructions. Examples of control flow instructions are jump, call and branch. These instructions have static and dynamic variants. For the static variant, a destination is attached in an instruction. For the dynamic variant, a destination is known at run-time. The most fundamental dynamic control flow instruction is branch, so it becomes the name of *BB*.

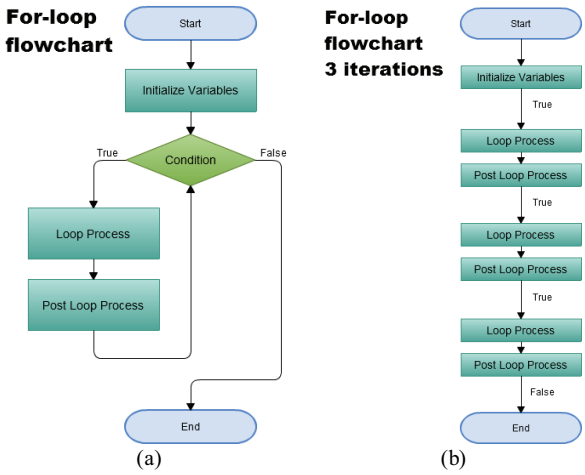

Fig. 3. is a flow representation of for-loop program (b) is an expansion of (a) by substituting branch decisions

To introduce of tracing dynamic control flow instructions, Fig. 3 (a) shows a for-loop flowchart. The key element in this flowchart is the green decision block or branching. The loop can be represented by decisions made in the green block. *BB* records these decisions. With a known flowchart (a program) and made decisions, the previous execution is revealed as in Fig. 3 (b).

Fig. 3 (b) shows that *BB* is capable of performing perfect reconstruction, but it must collect all dynamics without fail to do this. Otherwise, incomplete traces will lead to a completely wrong execution path. *BB* records register/memory used in jumps, register/memory used in calls and branches. These are complete to trace a bare-metal system. In a system with operating system, there are more dynamics. They influence or interfere a program execution's path from outside the program's code. They are hardware/software interrupts. The way these behave is defined by either hardware or an operating system. *BB* must know these dynamics too.

If a system is capable of multitasking (which is common in computers these days), *BB* has to take a note which task is switched from and to too. An identifier either an instruction or a pattern is needed to clarify this process.

Traces are recorded in a packet, and the packet is fired once it is closed. There are two conditions to close a packet: a packet is full and a flow disconnects e.g. an interrupt happens. A packet also has a header to ease trace processing. The information inside a header is the number of branches, packet start address, reason to start a new packet and other supplemental data. *BB* requires memory only for a single packet and some other states for a header. A packet is handled by another module depending on how developers want to utilize traces.

BB differs from *perf branch* in Section II in these following features:

TABLE I. PERF BRANCH AND BRANCH BITSTREAM DIFFERENCES

	perf branch	***BB***
Record	branch	all dynamics
Trace size of a branch	12 bytes (32-bit), 24 bytes (64-bit)	1 bit
Trace collection	make an interrupt to kernel	handle by another module

B. Processing Traces

A *trace-driven simulator* as in [3] is used to process *BB* traces to perform perfect reconstruction as in Fig. 3 (b). [3] relies on *Program Trace Graph* (*PTG*) concept. *PTG* is a derived version of *Control Flow Graph* (*CFG*). *CFG* is a graph representation of possible execution paths inside a program. *PTG* groups sequential non-flow controlling instructions and a controlling instruction into a single node.

Fig. 4 (a) shows a *CFG* of a function calling another function. Non-flow controlling instructions, represented as blank rectangles, are grouped with the closest flow controlling instruction. *CFG* from Fig. 4 (a) is transformed into *PTG* in Fig. 4 (b). Once a *PTG* of a program is obtained, a reconstruction is straight forward by using *BB* traces. Another thing that it has to do is keeping track of a call stack.

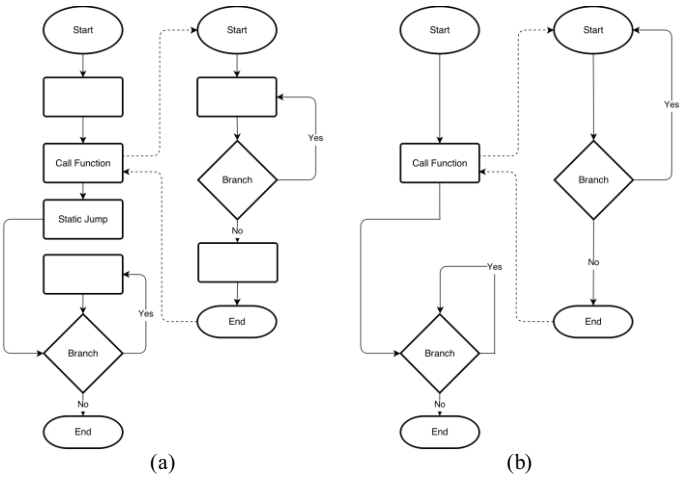

Fig. 4. a sample *CFG* is derived into *PTG* from in (b)

PTG is capable of modelling flows inside a program. However, a trace-driven simulator must be able to process external interferences, such as, system calls, exceptions or context switching, too.

For system calls, exceptions and interrupt-alike, these events go to finish some routines and return. [3] handles by going to that event and save a node to return to. For example, an exception occurred in Fig. 5 (a), [3] goes to its routine and return back to its original path as in Fig. 5 (b).

For context switches (switching tasks), these events swap call stacks. [3] tracks by using loaded stack address and saved stack address as context IDs.

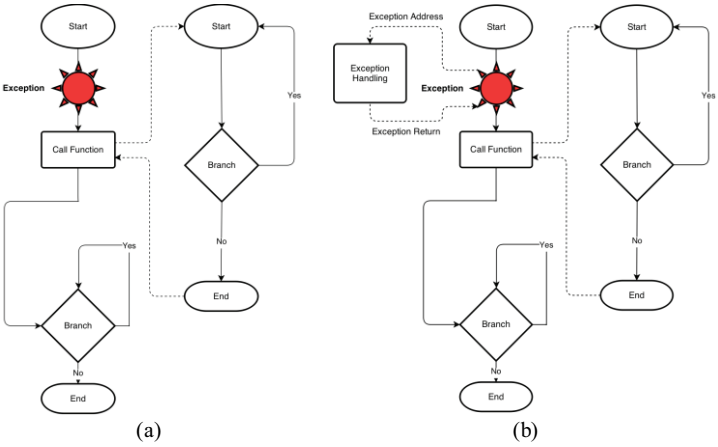

Fig. 5. an exception is recorded in a trace file (b) models this event as short circuit to its destination address

IV. QEMU IMPLEMENTATION

QEMU is a generic open-source machine emulator and virtualizer. *QEMU* can be run on and can emulate many CPU architectures. It has gained attraction lately as emulators for Android, Ubuntu phone and Sailfish OS in recent years.

QEMU creates *Tiny Code Generator* (*TCG*) to translate an instruction from one architecture to another. It uses *TCG Intermediate Representation* (*TCG IR*). It is a *RISC* instruction set. It also provides register and temporary variable creations and labelling inside *TCG IR*. Fig. 6 shows the translation process. An instruction from one architecture is translated into *TCG IR*. The *TCG IR* may consist of several *TCG* instructions. After that, a *TCG* instruction is then translated into instructions in another architecture. It accepts an instruction from a binary of an architecture to be emulated.

Fig. 6. Code translation in *QEMU*

However, instruction translation into *TCG IR* is sometimes tedious. Instead of trying to map an instruction to the limited set of *TCG* instructions, there is a *TCG* instruction allowing a translation to call user-defined C functions to aid the translation process. This C function is called a *helper function*.

BB is implemented as an extension to *QEMU* 0.14 for *TCT* architecture [6]. It takes the advantage of *helper function*. Instead of using it to aid instruction translation, it is used to perform tracing routines. Since tracing routines do not affect emulated CPU's states, the emulated machine does not notice being traced. Instrumentations were done in events in Table II.

TABLE II. INSTRUMENTED EVENTS

Intra-program dynamic control flow events	Inter-program dynamic control flow events	
branch	exception	reset
register branch		system call
register jump		CPU
register call		MMU
		IRQ
	return from exception	
	context switch	

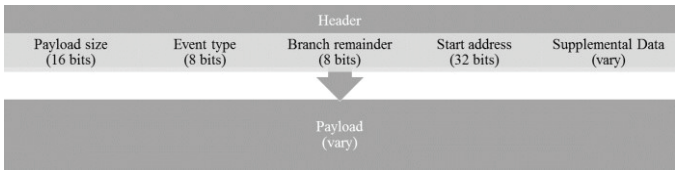

Fig. 7. *Branch Bitstream* packet structure

A packet structure is shown in Fig. 7. A header contains payload size, event type, branch remainder, start address and supplemental information to that type. The unit of payload size is byte. An event type serves as a reason starting its current packet. There are 3 event types: *normal*, *exception* and *context switch*. *Normal* type means continuation from a previous packet as control flow still happened explicitly in a binary. *Exception* type means it was force jumped by an event. There are 5 exception types in *TCT*: reset, system call, MMU, IRQ and CPU exception (computation error e.g. divide by zero). *Context switch* type means data context used by a process is changed by another process. This event can be noticed by inspecting an immediate offset value of memory store/load instructions. Supplemental data for each type is shown in Table III.

TABLE III. SUPPLEMENTAL DATA FOR EVENT TYPES

Type	Supplemental data		Size (bit)
BB_normal	None		0
BB_exception	Exception return address		32
BB_context_switch	Previous	Context stack address	32
		Program counter	32
	Next	Context stack address	32
		Program counter	32

The payload part contains branch decisions and destination addresses. A branch decision is represented by 1 bit. A destination address is a full address of that architecture. It is 32 bits for *TCT*. This then fills the previous *BB* packet header's payload size which is 2^{16}-1 bytes in this configuration.

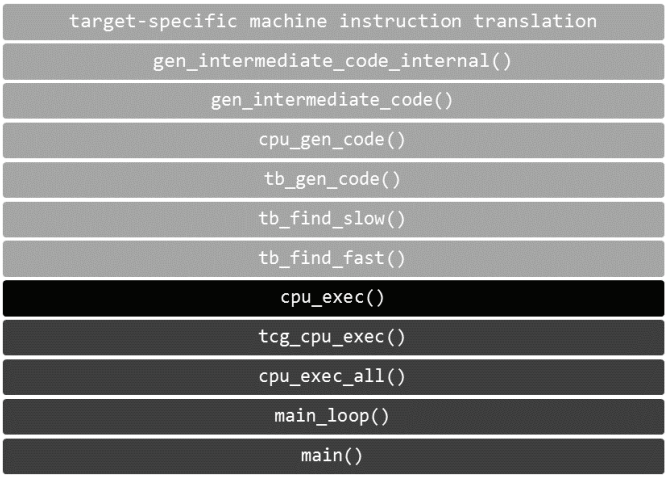

Fig. 8. *QEMU*'s call stack during binary translation

Opening and closing a *BB* trace file is handled at `cpu_exec()`.It is the furthest function that is architecture independent and is unique to one emulated CPU instance. According to Fig. 8, functions higher than `cpu_exec()` handle translation, and functions lower than `cpu_exec()` handle emulator states. `cpu_exec()` executes translated code and makes a decision upon an interrupt. It can decide to make an interrupt to be handled by an architecture defined routine or to be passed to `cpu_exec()`'s callers.

A packet is closed and written to the trace file when:

1. *QEMU* returns from `cpu_exec()`
2. Exception or context switch *BB* event type occurred

V. EXPERIMENT

Linux kernel (3.0.0-rc7) boot on a *Milkymist* board (modified to use *TCT* instead) was profiled in this work. Its booting was executed in *QEMU* with *BB* extension from Section IV. The environment of this experiment is as in Table IV, and the emulation overhead is showed in Table V. The *trace-driven simulator* [3] used the produced trace file and the kernel binary to reconstruct. After processed, the characteristic of a *BB* trace file and its boot profile are showed in Table VI.

TABLE IV. EMULATION SETUP

Emulation Target	
Target Binary	TCT ported Linux 3.0.0-rc7
Compiler	TCT toolchain version 5 tct-elf-gcc 4.5.2
C library	Newlib-1.19.0
Init Ramdisk	None
Kernel Arguments	None
QEMU Parameters	
QEMU version	0.14.50
QEMU enabled options	-singlestep -logfile -d in_asm,exec,count_insn
Board	TCT-Milkymist
TCT Clock	80 MHz
MMU	Enabled
Host Specification	
Host CPU	2x Intel Xeon E5630
Host RAM	16 GB
Host OS	RHEL 5.9

TABLE I. EMULATION OVERHEAD

	Without BB	With BB	Difference
AVG Time (s)	8.467	18.752	+10.285 (+121%)

TABLE II. LINUX KERNEL BOOT PROFILE

TRACE FILE			
BB data file size (MB)			7.0
Total packets			80,784
BB_normal **packets**			2,553
BB_exception **packets**			78,187
BB_context_switch **packets**			45
Minimum payload size (4-byte)			1
Maximum payload size (4-byte)			8,300
Average payload size (4-byte)			19.70
RECONSTRUCTION			
Average reconstruction time (s)			2.91
Calculated actual run time (s)			3.35
Total cycles			267,878,548
Effective total cycles (before panic)			234,585,351
Most cycle spent function			unpack_to_rootfs
Its spent cycles			167,772,522
Ratio to effective total cycles			71.52%
Most called function			strcmp
Number of times called			519,763
Ratio to effective total cycles			0.22%
Exception Handling (excluding after panic)	**Number of times called**	**Reset**	0
		System call	0
		Execution exception	0
		MMU exception	75,003
		Interrupt request	2,186
	Spent cycles	**Reset**	0
		System call	0
		Execution exception	0
		MMU exception	13,668,815
		Interrupt request	2,809,966
	Average spent cycles	**Reset**	0
		System call	0
		Execution exception	0
		MMU exception	182.24
		Interrupt request	1,285.44
	Ratio to effective total cycles	**Reset**	0
		System call	0
		Execution exception	0
		MMU exception	5.83%
		Interrupt request	1.20%

The 2 times emulation overhead in TABLE I causes by a large number of *BB_exception* events occurred. Many packets were closed and fired right away from this event type. Moreover, the average packet payload size is only 20 bytes. If they were all destination addresses, a packet contained only 5 addresses. Using I/O for one time adds large overhead. It is not efficient to use I/O frequently.

Hotspots inside a program can be queried out from a trace. 10 most time consuming functions and called functions are shown in Fig. 9 and Fig. 10, respectively. Most time consumed functions per call are shown in Fig. 11.

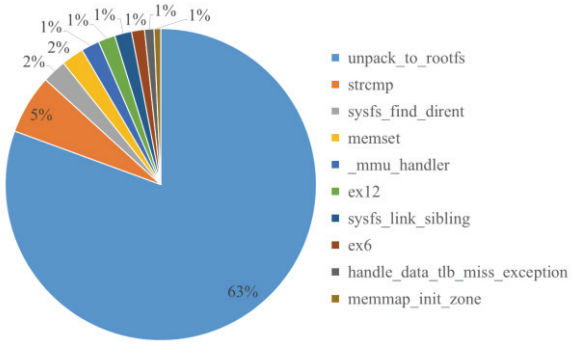

Fig. 9. Most time consuming functions

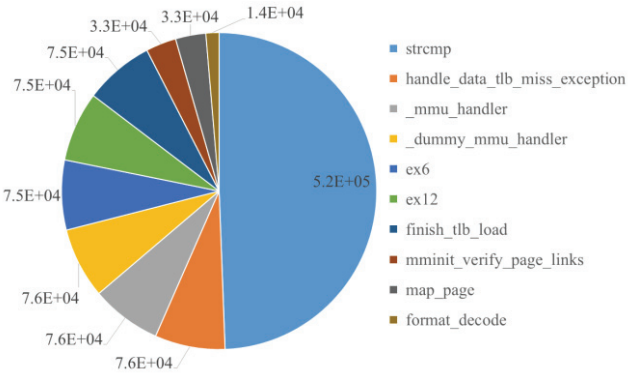

Fig. 10. Most called functions

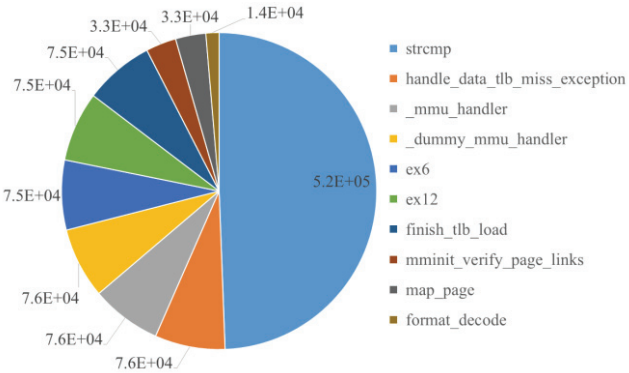

Fig. 11. Most time consuming functions per call

[3] offers CLI for manually stepping through CFG nodes. A snapshot is showed in Listing 2.

```
[Go to head BB-section]
(0:0: 0: )[     0] 000064 : [     3] (START_F_JUMP_TARGET_DEFAULT) : nop
(_start)
BB-navigate > s
[Step to next TRACE]
(0:0: 0: )[     4] 000064 : [     4] (START_F_JUMP_TARGET_DEFAULT) : nop
(_start)
BB-navigate > s
[Step to next TRACE]
(0:0: 0: )[       8] 000074 : [    193] (START_F_DEFAULT) : mts IDX,R3
(_invalidate)
BB-navigate > s
[Step to next TRACE]
(0:0: 1:1)[       9] 000084 : [    190] (BRANCH) : bnz 74 <_invalidate>
(_invalidate)
BB-navigate > c
[Step to next CALL (any level)]
(0:3: 0: )[     410] c000002c +: [       4] (CALL) : call c0003598
<machine_early_init> (_sinittext)
```

Listing 2. Trace navigation in [3] trace-driven simulator

VI. Discussion

The concept of perfect reconstruction is possible and is showed by profiling kernel boot in Section V. Nonetheless, an implementation of an execution environment in an emulator is not feasible. An emulator is still a software program. Migrating tracing routines to an emulator still causes overhead as in Fig. 1. *BB* is realistic when tracing and execution can run independently of each other. Thus, the concept should be implemented in hardware in practical uses.

If *BB* is implemented in hardware on the same chip as a CPU, these are 3 things to be considered: a combination of a CPU architecture and an operating system, performance degradation and available silicon area. The combination tells which events needed to be traced. Performance degradation is introduced by extra capacitance load to drive outputs of a CPU to *BB* and a time margin to let *BB* completes its tracing routines before the next one occurs. *BB* needs memory to store a payload and states, so more silicon area is used on a chip.

At the minimum, it would require these components:

1. Input pipe from a CPU
2. Output interface from *BB* module
3. Finite-state machine for tracing routines
4. Buffer space for a packet payload

It depends on a design where to map *BB* packet out port to. It can be mapped to a debugging port to another equipment, to a hard disk drive to collect traces for long runs or to another component to digest and provide feedbacks to CPU to improve performance.

At the current state, *BB* still lack features in real world applications. It has to tackle 3 more issues: trace compression, portability, and concurrency.

Firstly, the only compression it has is storing a branch decision in 1 bit instead of a full address. Run-time known destination addresses are recorded in full width. If the latter overwhelms the former, a trace file is barely compressed. It is still not favourable in always-on systems as in embedded applications.

Secondly, *BB* suffers a portability issue, because it has to know dynamic control-flow events of a system. A system means a pair of a CPU architecture and an operating system. This limits the use of *BB* from one product to another, even they are developed by the company.

Thirdly, concurrency measure has not been developed. Nowadays, embedded computers have many processing cores. Timestamp mechanism needs to be developed in order to coordinate the reconstruction.

VII. Conclusion

Tracing is an important method to software development. There are many tracing tools available, but they are mostly software-based. This means tracing routines intervene original execution of software. Embedded systems may not be tolerance to this intervention. *Branch Bitstream* does differently by migrating tracing routines to an execution environment. *BB* traces a system at machine instruction-level. The advantages of *BB* are retaining original behavior, capturing complete history and simple implementation. This concept was implemented in *QEMU* and experimented with Linux kernel boot. The booting was profiled and can be used for further inspection by manual stepping.

Acknowledgment

The first author thanks to Methavanitpong family for moral support and MEXT for providing scholarship during the study.

References

[1] M. Ekman and H. Thane, “Software Instrunmentation of Safety Critical Embedded Systems - A Problem Statement,” in System, Software, SoC and Silicon Debug Conference, Vienna, 2012.

[2] J. R. Larus, “Abstract Execution: A technique for efficiently tracing programs,” University of Wisconsin-Madison, Madison, 1990.

[3] T. Isshiki, D. Li, H. Kunieda, T. Isomura and K. Satou, “Trace-driven workload simulation method for Multiprocessor System-On-Chips,” in Proceeding DAC '09 Proceedings of the 46th Annual Design Automation Conference , New York, 2009.

[4] A. Nagai, “Introduce New Branch Tracer 'perf branch'”, in LinuxCon Japan, Kanagawa, 2011.

[5] Intel, Intel 64 and IA-32 Architectures - Software Developer's Manual Volume 3, 2014.

[6] T. Isshiki, D. Li and H. Kunieda, “Multiprocessor SoC design framework on Tightly-Coupled Thread model,” in 2008 International SoC Design Conference, Busan, 2008.