Efficient Function Call Tracing with Link-Time Binary Rewriting for CE Devices

Bon-Keun Seo, Jinkyu Jeong, Joonwon Lee, and Euiseong Seo

**Abstract — ***As the scale and complexity of software components in consumer electronics increase, the importance of performance optimization is rapidly growing. Consequently, the demand for performance optimization tools tailored for the consumer electronics environment is stronger than ever. A function call tracer is a vital tool for investigating relationships between functions, invocation counts of a function, and elapsed time in a function. Despite its importance in performance optimization, the limited capability of embedded hardware prohibits use of existing dynamic binary instrumentation tools. Moreover, the use of closed proprietary components excludes source-level analysis tools out of viable options. In this paper, LITIFUT, a function call tracer designed for consumer electronics, is proposed. This tool rewrites an executable file or a library file to inject profiling code during the linking stage. This approach achieves as little performance overhead as source-level instrumentation and as minor developer-intervention as binary-level instrumentation. The prototype implementation supports the two most popular embedded processor architectures. The evaluation with a real-world embedded software application showed that LITIFUT successfully profiles program activities with insignificant overhead.* [1]

Index Terms — Embedded systems, Function call tracing, Binary rewrite, Static instrumentation

I. Introduction

For most embedded software developers, function call traces are invaluable data for both performance optimization and debugging. Function call traces reveal the internal activities of software, elapsed time in each function, causes of abrupt system crashes, and so on. Therefore, function call tracers are being used as veritable stethoscopes by software developers.

Function call tracers are also useful for consumer electronics development. For example, these tools can be used to reduce the booting time of digital TVs, smartphones, and other devices [1], [2]. For this purpose, a developer collects function call traces during the booting stage and determines which function calls can be delayed to a post-boot time. The functions can then be removed from the boot process. Function call tracers are also helpful in identifying the sum of time taken for each function and in count the number of function calls [3]. Obviously, the function call log is aggregated by the function to yield the sum or the count. This information is the most useful for optimizing embedded software performance. In addition, when a consumer electronic device must customize the kernel or system libraries, such as the buffer cache [2], [4], function call traces operate as a strong analysis tool for understanding and comparing the behavior of the customized algorithm.

Although many function call tracers based on diverse methods have been developed, they share a common requirement: the target application must be run at least once to obtain the results. The runtime overhead that a tracer tool incurs becomes a significant burden when the software to analyze is running on consumer electronic devices with limited computing capabilities. In fact, the overhead may cause the device to malfunction or to run too slowly to trace.

A compiler level tool [5] provides an option to insert function call trace stubs for the entry and exit of all function calls; it can therefore be used as a Swiss-army knife for function call analysis. However, source-level instrumentation is not suitable for most cases in the CE device development process. Out-sourced proprietary libraries that are typically used in these devices do not allow access to source code. Because it requires source code, some functions in these libraries cannot be traced. Moreover, all the code in the software application must be compiled using a specific option that should not be used in the final product. Because source code compilation of consumer electronic devices typically takes hours, it impedes the performance of the software developer.

Function call tracing can be implemented using a binary instrumentation tool [6]–[14]. This type of tools has significant runtime overhead because binary instrumentation is usually implemented using runtime binary translation or runtime probe injection. The runtime overhead makes the binary instrumentation technique impractical for CE devices, which are very slow even without the instrumentation. Most of this overhead is a result of the generality of binary instrumentation tools, which enables deeper analysis in addition to function call tracing.

To address the above issues, LITIFUT, the acronym of *Link-Time Function Tracer*, which is a link-time binary rewrite tool for function call tracing in CE devices, is proposed. Because it performs function call tracing only, LITIFUT minimizes runtime overhead while adding the

[1] This work was supported by the IT R&D program of MKE/KEIT [KI10041244, SmartTV 2.0 Software Platform] and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-0006423).

Bon-Keun Seo is with the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea (e-mail: bkseo@camars.kaist.ac.kr).

Jinkyu Jeong, Joonwon Lee and Euiseong Seo are with the Sungkyunkwan University, Suwon, Korea (e-mail: {jinkyu,joonwon,euiseong}@skku.edu).

Contributed Paper
Manuscript received 10/02/13
Current version published 12/24/13
Electronic version published 12/24/13.

convenience of binary instrumentation tools. An embedded software application applying LITIFUT runs as fast as one without it. In addition, it is very easy to apply LITIFUT. A developer easily writes trace recording subroutines to fit for the specific development environment and for its own tracing purpose. The tracing option is turned on and off during link-time by simply setting an appropriate environment variable.

For our study, the prototype of LITIFUT is implemented for the two most popular embedded processor architectures. It was then evaluated with a real-world consumer electronics workload and two performance benchmarks.

The rest of this paper is organized as follows: Section II introduces the design principles of LITIFUT and Section III presents its implementation details. The profiling automation front-end of LITIFUT is presented in Section IV. Section V evaluates the prototype implementation. Finally, Section VI concludes the research.

II. Design

The design of LITIFUT is inspired by the probe injection mechanism used in dynamic binary instrumentation tools such as KProbes [12], SystemTap [13], DTrace [14], and so on, except that it is statically performed at link-time. With minimized runtime overhead, it eliminates the necessity of user intervention in source-level instrumentation by rewriting the binary file.

A. Goal

The desired properties of the proposed function call tracer are as follows:

- Because it is targeted for CE devices, it should support, but not be restricted to, embedded processors.
- It should be able to efficiently trace all functions in the binary file, except user-defined exceptions.
- It is better to have as little overhead as possible. The overhead accounts for both additional memory capacity and CPU cycle.
- It should be easy to use. Less user intervention is most desirable, even without build system modification.
- To enable deeper analysis, it should support dynamic shared libraries.

B. Architecture

LITIFUT rewrites a binary file at link-time. It performs this function by using a mechanism similar to probe injection, which substitutes a machine instruction to call a probe that emulates the original instruction, while also executing the additional job that the probe was meant to perform.

Unlike dynamic instrumentation schemes, LITIFUT statically inserts a tracer module into the binary. It makes the tracer efficient and able to trace from the beginning of the execution. For this purpose, LITIFUT uses the linker to link the tracer module object files in addition to the original object files. If the source object files are not available, a disassembly and reassembly technique [15], other than using the linker, is applicable. However, LITIFUT assumes that the object files are always available because the function call tracers are most valuable in development phases.

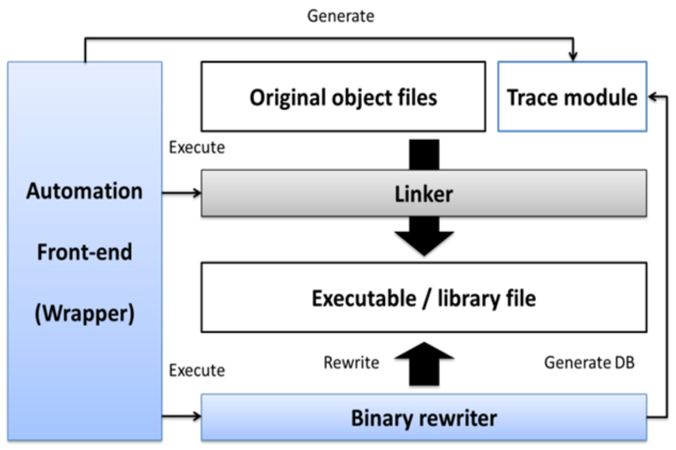


Fig. 1. The architecture of LITIFUT. The binary rewriter, the tracer module, and the automation front-end are the building blocks of LITIFUT. The front-end uses the linker to produce a binary file to rewrite from the original object files.

To instrument function calls, LITIFUT modifies existing function calls to instead jump to the tracer module, which records and emulates each function call on runtime. It is the binary rewriter's job to overwrite function call instructions in executable and library files. The rewriter also records the overwritten function call instructions for the tracer module to emulate them on runtime. The binary rewriter and tracer module must be architecture-dependent to be able to correctly rewrite and emulate the function call machine instructions.

For ease of use, LITIFUT provides an automation front-end, which works as the wrapper of a compiler toolchain and simplifies the process of linking the tracer module and rewriting the binaries. Using this front-end, a developer can easily enable and disable the function call tracer by simply turning on and off a specific environment variable.

The overall architecture of LITIFUT is illustrated in Fig. 1. The figure depicts how each module interacts to trace function calls in CE devices.

III. Binary Rewriting

The binary rewriter is designed based on function call application binary interfaces (ABIs) for embedded processors [16], [17]. The binary rewriter substitutes each function call instruction that is a 4-byte machine instruction, into one jump instruction that is directed to the tracer module. Because one machine instruction is replaced by another, no additional address adjustment is needed; therefore no update to, even existence of, the relocation information is required.

For the instruction overwrite, the rewriter should have the following mechanisms for each processor type. First, the rewriter should be able to differentiate instructions from data in the binary file. Second, the rewriter should have a mechanism to filter function call instructions from all instructions in a software application. Finally, the rewriter should choose one machine instruction that jumps to the tracer module without relocation.

TABLE I
EMBEDDED PROCESSOR FUNCTION CALL INSTRUCTIONS

Processor	Instruction	Type	Next PC	Usage
EMBProc A	bl / blx (I)	Immediate	PC + Immediate	Static functions
	blx (R)	Register	Value of R	Virtual functions
	ldr $PC, (R)	Memory	Value of address R	PLT, veneer functions
EMBProc B	jal (I)	Immediate	$PC_{31..28} \oplus I_{25..0} \oplus 00$	Static functions
	jalr (R)	Register	Value of R	Virtual functions
	jr (R)	Memory	Value of R	PLT

A. *Function call application binary interfaces*

For each processor and runtime environment, the function call ABI is defined for compatibility. With them, various compilers from various languages can cooperate to produce a working binary file. The ABI constitutes a few function call standards. It defines the machine instruction to use for a certain circumstance, the parameter and the return value passing standards, the use of stack frames, and the register usages. When a compiler is compatible with the ABI, there must be a limited number of function call patterns in the executable files. Therefore, one can assume that there is no function call that is out of that pattern when it is built using such a compiler.

In an embedded processor, henceforth called EMBProc A [16], function calls are implemented using the instructions outlined in Table I. Each of these instructions uses an immediate value, a register, or a value stored in memory as a reference to the target function address. After the function call, the return address is stored in the `lr` register, and the `sp` register should reference an appropriate stack address to store local variables, and so on. Function parameters are passed using `r0-r3` registers and the stack, and the return value is set to the `r0` register.

The compiler for EMBProc A translates static function calls using the `bl` instruction, which relatively jumps to a location within the 64MB boundary. Therefore, a static function call cannot jump to a function that is located beyond the boundary. Whenever the linker detects that the function call exceeds the 64MB boundary, it generates a veneer function that bridges the gap between caller and callee. A veneer function indirectly jumps to the destined function using an `ldr $PC, [$PC-4]` instruction with the target address stored beside the `ldr` instruction. The long-distant function call is then replaced by a call for veneer function. Because the veneer function does not change all the registers including the `lr` register but the `pc` register, the callee correctly runs and returns to the caller without any modification.

An `ldr` instruction is also used for the procedure linkage table (PLT) to indirectly call dynamic shared library functions. However, it loads the address in the global offset table (GOT) in place of the address stored beside it.

Function call instructions in the other processor—here, EMBProc B [17]—are simpler than those of EMBProc A. There are only two instructions in use and they are absolute jump instructions. Although there are additional relative branch instructions, they have a 256 KB addressing limitation because only 16 bits of the instructions are used for an immediate. Therefore, they are not suitable for generic function calls because they complicate the compiler and linker implementation. The return address should be stored in the `ra` register, and the `sp` register holds the stack pointer. Parameters are stored in `s0-s9` registers and in the stack, and the return value is stored in `v0` and `v1` registers. The `jr` instruction is used for the PLT function call, similar to what the `ldr` instruction does in EMBProc A. In most operating systems, the `jalr` instruction uses only the `t9` register for function address holding, and `t7` and `t8` registers are reserved for the dynamic linker.

B. *Binary rewriter*

The binary rewriter overwrites all function call instructions in the binary into the call for the tracer module. The overall operation of the binary rewriter is illustrated in Fig. 2. At first, the rewriter filters out the function call instructions. For this purpose, it segregates instructions from the data and checks whether or not each instruction is a function call instruction. All the instructions are stored in `.text`, `.init`, and `.fini` sections. However, there might be read only data in the code sections, because some compiler optimization algorithms store function static constant variables within the instructions of the function. In this case, the compilers for EMBProc A generate mapping symbols such as `$a`, `$d`, and `$t`, to mark regions for instructions, data, and thumb instructions, respectively [16]. In the case of the compilers for EMBProc B, each symbol table entry for functions in the ELF file contains an (address, size) pair of the range wherein the instructions are stored.

Next, the rewriter checks whether or not each instruction is a function call. The identification is performed based on function call ABIs for the specific architecture. LITIFUT rewrites instructions for static function calls and virtual function calls only, not for PLT and veneer functions. This is because PLT and veneer function calls can easily be post-processed after the application execution.

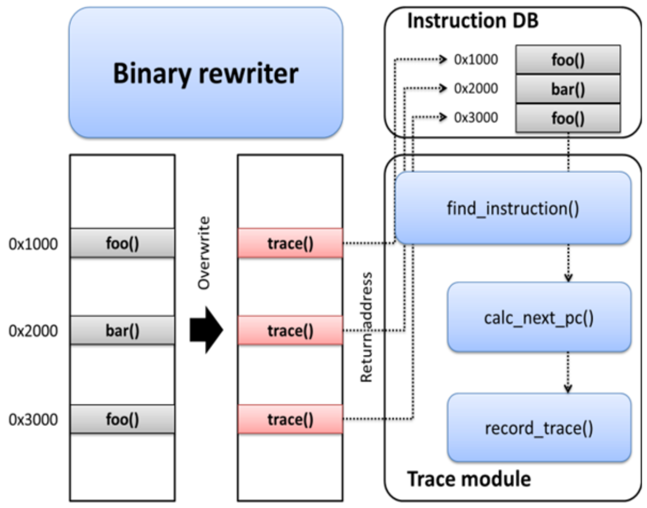


Fig. 2. Operation of binary rewriter and tracer module. The binary rewriter substitutes all function calls to instead jump to the tracer module, which in turn appropriately emulates and records the function call.

Finally, the function call is substituted by a replacement instruction. In EMBProc A, the only instruction that directly jumps to the tracer module is the `bl` instruction. Because the `bl` instruction cannot jump beyond the 64 MB range, the assembly stub of the tracer module must be repeatedly inserted to be in the vicinity of the `bl` instruction. This is the job of the automation front-end, as explained in Section IV.

In EMBProc B, the `jal` instruction performs the same role as the `bl` instruction in EMBProc A. Unfortunately `jal` is an absolute jump instruction, so that special care is required when it rewrites the dynamic shared libraries (See Section III-F).

While the function call instruction is replaced, the original instructions are gathered in an instruction DB. Each record in the DB is a pair of (address, instruction). The instruction DB is used to emulate the function call in the tracer module, as explained in Section III-C.

C. *Tracer module*

The tracer module emulates the original instruction to jump to the appropriate function after it records the function call in the log. On runtime, every function call jumps to the tracer module. The control flow first reaches the assembly stub, which saves all the registers, calls the routines recording function call traces and emulating function calls, and finally jumps to the original target function. The assembly stub implementation detail is architecture-dependent.

In EMBProc A, the stub jumps to the originally intended function by setting the `pc` register to the emulated `next_pc` value. The emulation is performed by calling `calc_next_pc()` function with the saved instruction as an argument. From the many ways of manipulating the `pc` register, the assembly stub uses the method of modifying the register value saved in the stack before restoring it at the end of the stub. In this way, all registers except the `pc` register hold identical values before and after the stub execution.

Algorithm 1. Assembly stub for the tracer module

```
procedure embproc_a_stub()
    stack ← push(registers)
    instr ← find_instruction($lr)
    next_pc ← calc_next_pc(instr)
    record_trace(next_pc)
    stack[pc_offset] ← next pc
    registers ← restore(stack)
procedure embproc_b_jal_stub()
    stack ← push(registers - $t9)
    next_pc ← find_instruction($ra)
    record trace(next_pc)
    $t9 ← next_pc
    (registers - $t9) ← restore(stack)
    jr $t9
procedure embproc_b_jalr_stub()
    stack ← push(registers)
    record_trace($t9)
    registers ← restore(stack)
    jr $t9
```

Searching for the saved instruction from the instruction DB is implemented as a binary search function `find_instruction()` using the return address (`lr` register) as a key.

In EMBProc B, two assembly stubs are implemented: one for `jal;` the other for `jalr`. Because the `jalr` instruction always uses the `t9` register to hold the address of the callee, using a special stub eliminates the need for saving the `jalr` instructions, which thereby reduces the memory requirement for the instruction DB.

The `jal` stub performs the same way as the one for EMBProc A, except it directly uses the address. Because there is only one type of instruction that holds the target address as an immediate, the emulation can be conducted at link-time, which releases the burden of runtime emulation overhead.

The `jalr` stub is the simplest of all the stubs. It only saves some registers modified by the stub, calls the `record_trace()` function to make a function call log, restores the registers, and jumps to the function that the `t9` register references.

The tracer module must be loaded before the binary file begins running. Therefore, the module is statically loaded in the binary file. LITIFUT uses the static linker by modifying the linker command line to link the object files of the tracer module.

D. *Two pass scheme*

The binary rewriter creates the instruction DB as a by-product of its operation. The instruction DB must be loaded before the tracer module emulates the function call for the first time. Without the DB, the emulation fails and, the program cannot proceed. There are many options wherein the DB is stored, such as in a raw file, a DBMS, or the binary file itself. LITIFUT stores the DB in the binary itself, because it can eliminate any dependency for the DB loading; the DB is present in the address space whenever the binary file executes.

The instruction DB is built as an array of (return address, instruction) in a C source code file. It must be compiled and linked to the binary along with the original object files and the tracer module. However, the DB is created after the binary rewriter has finished running. Therefore, it is required to run the linker and the rewriter twice. The first time, they build the DB; the second time, they rewrite the binary.

The two-pass rewrite scheme, which is regarded as complex, can be reduced to a one-pass scheme using other binary rewriting tools [15]. However, it is simplified using the automation front-end, as explained in Section IV.

E. Trace recording

The tracer module supports various trace recording methods. If the developer determines what to record and where to store it, the module attempts to minimize overhead and user intervention. The developer can modify the `record_trace()` function to alter the behavior of the module.

The tracer module supports the following functions by default. First, it can be configured to copy the trace file to a USB thumb drive when the trace stops. Because recent CE devices typically support USB thumb drives, the automated copy operation simplifies the process of transferring log files for analysis.

Second, it supports a bulk-write option. By writing the traces in a large bulk, the tracer greatly reduces I/O overhead. However, this requires a thread-safe algorithm to correctly operate.

Third, it supports multi-threaded applications. Because a lock-free atomicity-guaranteeing algorithm guards the recording process, it can safely record multi-core, multi-threaded application traces. Considering the trend that CE devices employ multi-core processors, thread safety is a fundamental requirement nowadays.

Last, it can be configured to de-duplicate the trace online using a bloom filter. Nevertheless, some CE devices have a storage device with scarce capacity if the operation of the device does not depend on it. To trace function calls for those devices, the trace size must be minimized to fit in free memory space. In that case, the bloom filter becomes useful because it is better than losing all traces, even though there is little possibility of missing some function calls.

Using the default trace recorder, CE software developers might do most of their function call trace analysis.

F. Dynamic shared library

A dynamic shared library file is dynamically loaded into a free virtual memory region unlike an executable file, which is always loaded into the specified address in the ELF header of the file. Therefore, the runtime address of the dynamic shared library always changes whenever the library is loaded. There have been efforts to make dynamic shared libraries load into fixed memory addresses [18]; however, it is not always guaranteed. Because of the dynamic nature of runtime addresses, a program linkage table (PLT) and global offset table (GOT) are devised to reference functions and variables in dynamic shared libraries [19].

The dynamicity of addresses also affects the implementation of the tracer module. The module uses the return address of a function call to search the instruction DB, however, the address changes every time the library loads. Therefore, the stub adjusts the return address by subtracting the base address where the library is loaded. The base address can be easily obtained by a simple calculation: *runtime_address - ELF_address*. For the calculation, the stub function address in the ELF file is stored when the library is rewritten.

In EMBProc B, where only absolute jump instructions are supported, an absolute jump instruction cannot directly reference a function in the library, because the address of the function is unpredictably determined at runtime. Therefore, the library function of EMBProc B is indirectly called by calculating the address from the frame pointer register. For this reason, all function calls in a shared library are implemented using `jalr`, the register jump instruction.

The `jal` instruction used for the rewriting likewise cannot directly call the tracer module if it is inserted into the library. Therefore, it tries to call the module in the executable; however, the 256MB address limitation prevents the function call from occurring. The library loads into the dynamic loader predetermined memory region (0x200000000x2FFFFFFF), and the address range lies too far from the module in the executable (0x00000000-0x0FFFFFFF).

Therefore, it is required to use a tiny bridging stub that bridges the gap between the library function calls and the tracer module in the executable. The stub indirectly calls the module in the executable to determine the address of the module, and it has a fixed address near libraries (0x20002000) because it is mmapped to the address by the executable. The function call in the library is substituted to a call for the bridging stub by the rewriter so the call redirects to the appropriate address. Mapping of the stub occurs the first time that the function in the executable is called. Because a function call inside a library occurs after at least one function is called by the executable, there is no possibility of a library function calling another function without the stub.

IV. AUTOMATION

It is difficult to use the binary rewriter, because the rewriter takes multiple steps with many hard-to-derive parameters. Such parameters include the address and size of a function that will be excluded from rewriting. To simplify the process, the automation front-end is devised. From the fact that it requires two linker invocations, the automation front-end replaces the linker command line to build rewritten binary files as if it is one of the feature of the linker. This attribute makes LITIFUT a link-time binary rewriting tool. After the trace file is created by running the rewritten application in the target CE device, the raw trace file must be post-processed to be human readable. The automated postprocessor creates final trace files by reorganizing function call traces for each executable file and library file. The automation front-end makes the binary rewriting and trace processing an easy task by simply enabling an environment variable and running a post-processor application.

A. Linker substitution wrapper

The automation front-end replaces the linker using symbolic links. A linker invocation instead executes the front-end, which automatically runs all the tasks a binary rewriter must perform. These tasks constitute, copying and compiling the tracer module, executing the modified linker command line that builds a binary file, including a tracer module, and rewriting the binary file.

An environment variable is used to notify whether the automation front-end should work as a linker or rewriter. If the environment variable is not defined, or if it has an invalid value, the front-end only runs the linker to produce the unmodified binary file.

Another task of the front-end is to resolve the symbol table. Given a list of functions, object files, and library files that should not be rewritten, the front-end finds out the list of code sections that all items in the list take. It then passes the list to the rewriter in the command line to ignore the code sections in the rewrite processing. The code sections that each item takes can be found from the linker map file, which is created by a special linker command line option. (For the GNU ld, it is the `-Map` option.)

B. Trace post-processor

The trace post-processor combines the trace files to produce a unified view of the function call traces. The post-processor solves the problems of the raw traces as follows. First, because the rewriter does not instrument the veneer function, the function call trace contains the address of the veneer function instead of the function that is called by the veneer function. To correctly understand the function call process, the full log should contain both the veneer function and redirected function.

Second, the developer cannot identify the runtime address of a dynamic shared library by the raw trace only. While the application executes, the Linux kernel exports the `/proc/(pid)/maps` file that contains all the memory mapping information. From the file, the address where each library file is loaded can be easily resolved. Therefore, the tracer module copies the `/proc/(pid)/maps` file alongside the trace files when the trace stops and the trace file becomes ready.

Third, it contains the address of a PLT entry for each of the dynamic shared library function calls. The target function address can be found in GOT, however, it can be retrieved only on runtime. Nevertheless, it is possible to consult the symbol table for the GOT entry that is in a one-to-one mapping relationship with the PLT entry. Because the symbol (a function name) is more human readable than the address itself, it is sufficient to log it as ‘a function called through a PLT entry’.

Last, more than one process must be simultaneously analyzed. Because more than one application can share a dynamic shared library, a combined shared library trace is required if one desires to analyze the behavior of the library. Additionally, two or more applications can interact using the IPC mechanism. For this purpose, the post-processor is designed to load multiple application traces at once, and it is able to produce a combined function call trace to understand the inter-operations.

The trace post-processor implements all these features so that a useful human readable trace output can be produced from the raw trace files. The basic output is a series of names of functions called. For more detailed analysis, one can add an additional post-processor module to it.

V. Evaluation

All instrumentation tools accompany the observer effect that affects the operation of the instrumented software. In CE devices, the slowdown of the software can result in a malfunction of the device, particularly timing-sensitive devices (i.e., I2C devices). Moreover, the additional memory demand can result in a memory shortage, which causes the whole system to misbehave. Finally, the trace I/O matters because it can slow the software performance and waste scarce storage resources.

In this paper, measurements are carried out on two smart digital TV (DTV) platforms, one with a dual-core EMBProc A processor, and the other with a single-core EMBProc B processor. The DTV with EMBProc A has 1GB of RAM, 62MB of free internal storage space, and three USB ports to connect USB devices such as a USB thumb drive. The DTV with EMBProc B has weaker computing power, with 512MB of RAM, 30MB of empty storage capacity, and only one USB port.

To evaluate the performance of LITIFUT, three types of workloads are used on both architectures. First, the CPU intensive workload that serially runs numeric sorts, string sorts, bit manipulations, float-point manipulations, Fourier coefficient calculations, Huffman compressions, IDEA encryptions, and neural network simulations. Second, the I/O intensive workload that performs random file system reads and writes on a file. Last, a DTV application is used as a compound real-world workload. Properties of these workloads are listed in Table II. The DTV application for EMBProc B has less functional features than the one for EMBProc A. It is because that platform is used for low-end DTVs to lower the price. In addition, executable files for the CPU-intensive workload and the I/O-intensive workload has different numbers of function calls for both architectures, because static libraries linked to the benchmark application are not identical.

A. Memory requirement

The increase of the binary file size by adding the tracer module is the main cause of memory overhead. This could be easily measured by the size of the tracer module. The tracer module size is dependent on the number of function call instructions, because the DB comprises most portion of the module. This is analyzed in Table III.

TABLE II
PROPERTIES OF WORKLOADS USED FOR EVALUATION

	EMBProc A				EMBProc B			
	CPU intensive	I/O intensive	DTV executable	DTV library	CPU intensive	I/O intensive	DTV executable	DTV library
Binary size (bytes)	669,256	401,335	109,752,492	5,864,948	883,460	416,996	91,411,476	2,404,460
Functions	1,081	179	128,434	8,484	1,081	179	97,341	2,107
Static calls	1,063	5,207	1,210,466	77,777	4,927	6,995	687,101	0
PLT / Virtual calls	33	2	111,339	722	14	2	218,410	31,163

TABLE III
MEMORY OVERHEAD OF LITIFUT COMPARED TO SOURCE-LEVEL TRACING

	EMBProc A				EMBProc B			
	CPU intensive	I/O intensive	DTV executable	DTV library	CPU intensive	I/O intensive	DTV executable	DTV library
DB size	9,328	41,672	10,574,440	627,992	40,056	55,960	5,496,808	0
Module size	11,464	11,464	11,464	11,464	9,984	9,984	9,984	4,096
Source-level	34,592	5,728	4,109,888	271,488	34,592	5,728	3,114,912	67,424

The tracer module size becomes almost 10% of the original application in EMBProc A, which is quite large compared to the EMBProc B, where it takes 5% in the executable file and close to 0% in the library file. The reason for this difference is as follows. First, the module for EMBProc B does not save the `jalr` instructions in the instruction DB. It dramatically decreases the DB size. Second, the library function calls are solely implemented by the `jalr` instruction. That makes the trace module size of the library almost zero by requiring only the tiny stub, which takes 848 bytes but is rounded up to a page because the `mmap()` system call works in page units.

The memory consumption of the function call tracer is comparable to that of the source-level instrumentation and dynamic binary instrumentation. The source-level instrumentation inserts function calls at the entry and exit of all functions, which uses 32 bytes for each function. This resulted in a total of approximately 4.1 MB for the DTV executable file of EMBProc A. It is almost 2.5 times larger in EMBProc A, but only 1.7 times larger in EMBProc B. However, for the CPU-intensive workload, LITIFUT is more efficient, because there is almost the same number of functions compared to function calls. This memory overhead is tolerable, considering the advantages of LITIFUT, such as the possibility of tracing functions without source code or of tracing functions in dynamic shared libraries. Because the dynamic binary instrumentation uses a data structure similar to LITIFUT, the memory consumption is larger than or (in the best case) equal to that of LITIFUT. Additionally, it carries the runtime overhead of rewriting instructions at runtime.

B. *Performance overhead*

Performance degradation induced by tracing affects the correct operation of the device, and the user experience, while function calls are traced. The application execution time is measured to investigate the efficiency of LITIFUT, as depict in Fig. 3. The measurement is conducted in three configurations: without tracing (no trace), with tracing but without bulk write (no bulk), with tracing and bulk write (bulk write). Because the CPU intensive workload repeatedly runs a set of benchmark algorithms for a specified time, the execution time is calculated to the length of time required to run the string sort benchmark 1,000 times. Without bulk write, each function call writes its log to the trace file, which issues the system call each time. The DTV workload did not work correctly, which produced a number of error messages, because it has many timing-aware function calls interacting with I2C devices in the DTV platform. In addition, The CPU intensive workload runs indefinitely, more than an hour, so that it is not measured. The bulk write scheme solves this problem but increases the memory usage by the size of the buffer, which is a negligible overhead.

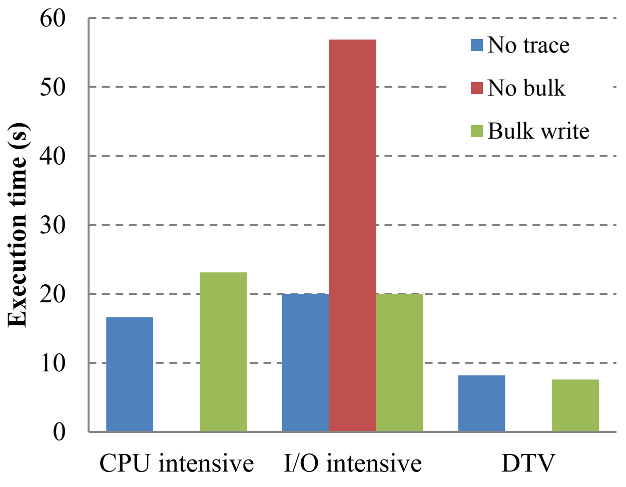


Fig. 3. Execution time overhead of LITIFUT. With bulk write, LITIFUT exhibits almost no overhead for I/O intensive and compound workloads.

With bulk writing, the execution time increases by approximately 40% for the CPU-intensive workload because it intensely performs function calls without idle time. This is one of the worst-case scenarios, which therefore proves the efficiency of LITIFUT. The I/O-intensive workload and DTV workload represent that the CPU overhead can be hidden by the idle time of the application.

C. Storage consumption

The trace file consumes storage space. The size of a trace file is dependent on the number of function calls, which in turn depends on the application execution time. Therefore, a trace file size can grow indefinitely, evading the precious storage capacity. In the DTV platform used for the evaluation, the internal storage capacity is very small, with only 62MB free, such that the trace file size has to be minimized.

Using the de-duplication feature of LITIFUT, the size of the trace file is minimized as shown in Table IV. Because the de-duplication feature can be used only for aggregation analysis, the sequence analysis workload cannot take advantage of it. Considering other tools cannot be applied even for aggregation analysis, due to the scarcity of the resources, LITIFUT becomes invaluable for the development of CE devices.

TABLE IV
TRACE FILE SIZE WITH / WITHOUT DE-DUPLICATION

	CPU intensive	I/O intensive	DTV
No dedup	65,921,024	16,862,656	4,734,976
Dedup	2,552	1,364	32,768

D. Instrumentation delay

Because LITIFUT repeats linking and rewriting process twice, it has a longer linking time than a normal link process. The instrumentation delay, however, can be disregarded in most cases because the target binary file is very small, whereby it takes less than a minute. The linking time of the DTV executable is exceptionally long, approximately 137 seconds on average. Therefore, the whole instrumentation delay takes approximately 5 minutes, including the binary rewriting time. Considering the compiling delay in the compiler-based method, and the runtime overhead in dynamic rewriting schemes, the instrumentation delay can be disregarded, which makes LITIFUT a viable solution.

VI. RELATED WORK

Because function call traces are very useful in software development processes, different tools have been developed to trace function calls in various ways.

Function call traces can be obtained using a compiler and a linker feature that adds instrumentation stub calls at function entry and exit [5]. In addition, Lattner et al. proposed a compiler-based platform for flexible instrumentation [21]. In that case, source code is required but is not always provided, particularly in embedded environments. Therefore, an efficient method to trace from binary files is needed.

A static binary rewriting method can be applied to add instrumentation calls in an executable file. For example, Smithson et al. proposed a novel binary rewriting method that disassembles a binary file and reassembles modified assembly code [15]. A more sophisticated binary rewrite tool [22] analyzes a binary file to discover program components and then inserts instrumentation stubs per instruction basis; however, it is built for a specific desktop environment, unlike LITIFUT, which works for CE devices.

The probe injection technique that inserts instrumentation routines during runtime using a breakpoint instruction can also be used to trace function calls in a running process [11], [12]. The instruction substitution process in LITIFUT is motivated by the probe injection procedure; however, LITIFUT has a better runtime performance than that method, because LITIFUT modifies the binary file while it is linked, not while it is running.

The dynamic rewriting scheme [5]-[9] rewrites pieces of a running program to alter the behavior of them and caches the rewritten instructions to avoid repetitive rewrite tasks. It also supports the function call tracing feature; however, its runtime processor and memory overhead are too heavy to be practical for CE devices with feeble processors and poor resources.

The LITIFUT approach is a hybrid solution combining aspects of the probe injection method and the static binary rewriting; LITIFUT uses a linker to insert the probe stub and a binary rewriter to substitute function call instructions. Therefore, it has a runtime processor and memory overhead that can be categorized between the compiler-based approach and static binary rewriting method.

VII. CONCLUSION

Despite the rapid improvement of embedded hardware, the even faster growth of software component complexity often induces a sluggish response in state-of-the-art consumer electronics, such as DTVs and smart phones. The larger the software code, the more improvement the code optimization will earn. Consequently, the role of software optimization in the development of CE devices is becoming crucial.

A function call tracer is one of the most quintessential performance optimization tools. However, existing function call tracers, which were primarily designed for PCs and servers, are not adequate for consumer electronics for various reasons.

In this paper, a function call tracer named LITIFUT is proposed for consumer electronics development. By employing the linking time binary rewriting scheme, LITIFUT performs fast under a limited amount of computational resources. In addition, it is able to analyze even when the source code of object files is unavailable.

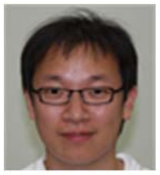
The prototype of LITIFUT is implemented for two popular embedded processor architectures, and it is evaluated with DTV operating software and two other benchmark suites. The evaluation showed that LITIFUT added an insignificant amount of execution time to the DTV software and only up to 10% of memory usage. The de-duplication feature of LITIFUT saved approximately 99% of storage space for collected data. Based on these results, it is concluded that LITIFUT is perfectly suited for the CE devices.

REFERENCES

[1] H. Jo *et al.*, "Improving the startup time of digital TV," *IEEE Trans. Consum. Electron.*, vol. 55, no. 2, pp. 721-727, May 2009.
[2] H. Jo *et al.*, "Optimizing the startup time of embedded systems: a case study of digital TV," *IEEE Trans. Consum. Electron.*, vol. 55, no. 4, pp. 2242-2247, November 2009.
[3] S. Park *et al.*, "Development of behavior-profilers for multimedia consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 55, no. 4, pp. 1929-1935, November 2009.
[4] M. Lee *et al.*, "PABC: power-aware buffer cache management for low power consumption," *IEEE Trans. Comput.*, vol. 56, no. 4, pp. 488-501, April 2007.
[5] R. M. Stallman, *Using the GNU Compiler Collection (GCC)*. Free Software Found., Inc., 2010.
[6] C. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. 2005 ACM SIGPLAN Conf. Programming language design and implementation,* New York, NY, USA: ACM, 2005, pp. 190–200.
[7] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proc. 2007 ACM SIGPLAN Conf. Programming language design and implementation*, New York, NY, USA: ACM, 2007, pp. 89–100.
[8] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the ARM architecture," in *Proc. 2006 Int. Conf. Compilers, architecture and synthesis for embedded systems*, New York, NY, USA: ACM, 2006, pp. 261–270.
[9] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Performance Computing Applications*, vol. 14, no. 4, pp. 317-329, November 2000.
[10] G. Ravipati *et al.*, *Toward the deconstruction of DynInst*, Comput. Sci. Dept., Univ. Wisconsin, Madison, Tech. Rep., 2007.
[11] D. L. Bruening, "Efficient, transparent and comprehensive runtime code manipulation," Ph.D. dissertation, Dept. Elect. Eng. and Comp. Sci., Massachusetts Inst. of Technology, Cambridge, MA, 2004.
[12] W. E. Cohen, "Gaining insight into the linux kernel with KProbes," *RedHat Magazine*, March 2005.
[13] F. C. Eigler, "Problem solving with SystemTap," in *Proc. Ottawa Linux Symposium*, 2006.
[14] R. McDougall *et al.*, *Solaris performance and tools: DTrace and MDB techniques for Solaris 10 and Opensolaris*, Englewood Cliffs, NJ: Prentice-Hall, 2006.
[15] M. Smithson *et al.*, "Binary rewriting without relocation information," Univ. Maryland, Baltimore, MD, Tech. Rep., 2010.
[16] S. B. Furber, *ARM System Architecture*, Boston, MA: Addison-Wesley Longman, 1996.
[17] The Santa Cruz Operation and AT&T, *System V application binary interface: MIPS RISC processor supplement*, February 1996.
[18] C. Jung *et al.*, "Performance characterization of prelinking and preloading for embedded systems," in *Proc. 7th ACM & IEEE Int. Conf. Embedded software*, New York, NY, USA: ACM, 2007, pp. 213-220.
[19] J. R. Levine, *Linkers and Loaders,* San Francisco, CA: Morgan-Kaufmann, 1999.
[20] B. De Sutter *et al.*, "Link-time compaction and optimization of ARM executables," *ACM Trans. Embedded Computing Syst.*, vol. 6, no. 1, 2007.
[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2nd IEEE/ACM Int. Symposium on Code Generation and Optimization*, Palo Alto, CA, USA: IEEE, 2004, pp. 75–86.
[22] T. Romer *et al.*, "Instrumentation and optimization of Win32/Intel executables using Etch," in *Proc. USENIX Windows NT Workshop*, Seattle, WA, USA: USENIX, 1997, pp. 1–8.

BIOGRAPHIES

Bon-Keun Seo received the B.S. and the MS degree in computer science from the Korea Advanced Institute of Science and Technology. He is currently a Ph.D. candidate in the Computer Science Department, Korea Advanced Institute of Science and Technology. His current research interests include file system, operating system, virtualization, and embedded system.



Jinkyu Jeong received the B.S. degree from the Computer Science Department, Yonsei University in 2005 and Ph.D. degree in computer science from Korea Advanced Institute of Science and Technology in 2013. He is currently a post-doctoral researcher at Sungkyunkwan University. His current research interests include real-time system, operating system, virtualization, memory management, and embedded system.

Joonwon Lee received his B.S. degree in Computer Science from Seoul National University in 1983 and M.S. and Ph.D. degrees from the Georgia Institute of Technology in 1990 and 1991, respectively. He is currently a professor in Sungkyunkwan University (SKKU). Before joining SKKU, he was a professor at the Korea Advanced Institute of Science and Technology from 1992 to 2008. His current research interests include low power embedded systems, system software, and virtual machines.

Euiseong Seo received his BS, MS, and PhD degree in computer science from KAIST in 2000, 2002, and 2007, respectively. He is currently an assistant professor in college of ICE at Sungkyunkwan University, Korea. Before joining Sunkyunkwan University in 2012, he had been an assistant professor at UNIST, Korea from 2009 to 2012, and a research associate at the Pennsylvania State University from 2007 to 2009. His research interests are in power-aware computing, real-time systems, embedded systems, and virtualization.