Development of Behavior-Profilers for Multimedia Consumer Electronics

Seonyeong Park, Jinkyu Jeong, Heeseung Jo, Joonwon Lee, Euiseong Seo

Abstract — *In spite of the rapid improvement of hardware performance, debugging and optimization still remain as important procedures for developing consumer electronics embedded systems due to the manufacturing cost and the product quality. However, because the properties of consumer electronics systems are significantly different from the traditional computing systems, the required functionalities of behavior-profilers for the multimedia consumer electronics systems have to be newly defined. We analyze the desirable characteristics of the behavior profilers for multimedia consumer electronics systems and based on the analysis results we also implement a novel profiler tool set which consists of light-weight profiler components and remotely executed GUI client programs. The implemented profiler tool set is independent to the processor architecture and able to analyze the whole system layers from operating systems to functions inside user-level applications. The effectiveness of our tool set was verified by actually performing optimization of a commodity digital TV system.*[1]

Index Terms — Software Profiler, Performance Optimization, Debug, Multimedia Software

I. INTRODUCTION

The functions of multimedia consumer electronics such as portable media players, digital televisions and MP3 players are rapidly becoming diverse, which is a result from the trend of digital convergence. In addition, the demand for the high performance in those devices is expanding to process high quality multimedia data. Thus, to achieve these requirements, the embedded systems in those multimedia electronics employ standardized platforms including high performance general purpose processors similar to the desktop counterparts. As a result, the embedded systems employ general purpose embedded operating systems which are similar to the operating systems being used in personal computers [1].

Applying general purpose embedded operating systems to consumer electronics makes the development process of the embedded software look similar to that of the traditional software production. Thus, the existing tools for developing traditional software are commonly used in the development process of consumer electronics embedded systems. However, the existing tools are not perfect for developing multimedia consumer electronics.

The embedded systems for the multimedia consumer electronics generally have hardware with marginal performance due to their production cost, energy efficiency and form-factor. The limitation of processor performance will remain for a long time especially in many mobile electronics. Moreover the use of high definition video and the process of massive and complex data are rapidly increasing. Therefore, the importance of the performance optimization process that is specialized for consumer electronics embedded systems is getting bigger and bigger.

Based on Amdahl's law [2], the execution time of each program module should be identified first in the optimization process. Therefore, the performance profilers have been playing an important role in the performance optimization step in traditional software production. However, the tasks in the multimedia consumer electronics systems have different characteristics compared to the traditional tasks which have been the target of traditional profiler tools.

In addition to the performance optimization, the stability of the system is also important since correcting the flaws in consumer electronics after purchase could be a heavy economical loss as well as a technical challenge. Therefore, the stability of the systems should be maintained. The tools for tracking dynamic memory usage and system interfacing are also essential for the task.

In this paper we analyze the requirements that the profilers for multimedia consumer electronics systems should have. Also, we suggest the objective properties of our prototype profiler tool set based on the analysis. Finally, we introduce the prototype system and the implementation experience of the prototype performance profilers which firmly have the objective properties.

The prototype implementation has GUI remote clients, real time profiling features, low computational and memory overhead, architecture independency and thorough profiling coverage from kernel to user-level applications. We briefly introduce the case study of optimizing digital TV system with our prototype implementation.

The remainder of this paper is as follows; Section II analyzes the requirements and defines the design objectives of the target profiler tool set. Section III describes the system model and implementation issues of the prototype profilers along with the case study of using the implemented profiler tool set. Section IV introduces the existing profilers and

[1] S. Park, J. Jeong and H. Jo is with the Division of Computer Science , Korea Advanced Institute of Science and Technology, Daejeon, Korea (e-mail: {parksy, jinkyu, heesn}@calab.kaist.ac.kr)
J. Lee is with the School of Information and Communication Engineering, Sunkyunkwan University, Suwon, Korea(e-mail: joonwon@skku.edu)
E. Seo is with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, Korea (e-mail: euiseong@unist.ac.kr)

comparison with our profilers. Finally, we conclude our work in Section V.

II. Design Objectives

The tasks running in multimedia consumer electronics share some common properties, which are not seen from the tasks in the traditional computing systems. By exploring those differences, we find the desirable characteristics of the behavior-profilers for multimedia consumer electronics one by one.

A. Real-Time Profiling

The existing profilers are generally designed for analyzing the behavior of long-term tasks. Thus, most of them produce analysis results after the completion of the target tasks. Therefore, the analysis result only reflects the behavior during the entire life-cycle of the task, not the instant reactions.

However, in the target systems, programs run continuously and endlessly right after powering-on of the system. The system will stay at sleep state waiting for a user command most of time and the actual process of the user commands are usually finished in short time since the reaction to user input longer than few seconds is rare in ordinary consumer electronics.

This interactive property makes a profiling interval unit become the reaction to a user command not the entire execution cycle of the tasks. Therefore the objective profiler should be able to gather profiling data during a certain user-defined interval. And also the profiling interval should be easily defined by simple and fast user interfaces because when a certain event will occur is hard to be predicted in some systems.

The analysis should be completed right after collecting the data and also the results have to be shown to the users instantly. These will help the users identify the problem points quickly and easily even though many operations occur continually in cascaded way.

B. Covering All Components

The applications in the target embedded systems are getting complex and, as a result, state-of-the-art multimedia consumer electronics gadgets tend to employ hardware architecture similar to the traditional computer systems which consist of general purpose processors, general purpose embedded operating systems with standard APIs (application programming interfaces) and purpose-specific middleware layers and applications to overcome the complexity and to shorten the development cycles.

Different from the traditional computers, most tasks in the multimedia electronics systems frequently interact with the peripherals devices. Also, the performance of the target systems is highly dependent on the performance of peripheral devices. Thus, the targets of profiling in those systems have to include not only user-level tasks but also the kernel internals as well as the interfacing structures between kernel and user-level tasks, to identify the interaction between applications and hardware components.

Especially, profiling the behavior of kernel internals enables the analysis of the elapsed time in each device-driver function. Understanding the time consuming pattern of each operation helps to identify performance bottle-necks caused by peripheral devices as well because manipulating peripheral devices may become a primary reason of the long latencies in reaction to user commands. Therefore, the kernel-level profiling is essential for performance optimization of such systems for resolving the latencies.

In general purpose embedded operating systems, system call mechanism is used for bridging between operating system kernel and user-level tasks. Thus, to precisely analyze the use of peripheral devices, monitoring the system call usage information is also necessary as well.

Generally embedded systems equipped in the target devices run continuously as long as the system is being powered on or used. Consequently, the life length of the tasks in the system is sometimes long to few hours or even few days. In addition the operations in the target systems are repetitive in many cases.

For these reasons, if a memory leak exists in a function to be repetitively called, all the memory space of the system will be used up by the memory leak. Moreover, the limited memory size of the target systems will significantly quicken the wreck. To improve the stability of the target multimedia electronics, all the memory leaks must be identified and removed thoroughly. Therefore, the profiler tool set for the target systems should be able to monitor the dynamic memory usage

C. Remote Data Analysis

Due to the cost efficiency, the processors embedded in the target systems are expected to have the tightly matched performance only for the designated jobs and, as a result, there is little margin usually. Therefore, in the development stage the time-consuming work like compiling or manipulating filesystem is done in the development systems which are fast and powerful.

In comparison to the computational applications, in which few recursive or repetitive functions are the focus of optimization, the tasks in the target systems consist of a lot of short sequentially-executed code blocks because reacting to user commands and controlling peripheral devices are basically sequential jobs. Thus, tons of data will be generated in a short time during profiling. Not to mention processing the data, storing the data locally in the target system may be unfeasible in many cases due to the limited hardware resources. To resolve this problem, the objective profilers have to separate processing data from collecting data by using two separated systems. Also, processing and collecting data should be done simultaneously.

Also, the target systems usually have no standard user-interfaces such as displays, keyboards and mice. The existing solutions for consumer electronics system developers are using serial- or network-connected terminals as the controlling interfaces. However, both collecting data and analyzing them are complex and have many parameters to control. Therefore,

text-based interfaces are not sufficient for controlling them.

Graphical remote control interfaces will resolve this matter. Therefore, the profilers need to be controlled with the user-interface program in the remote systems connected with network.

D. Portability

The processor architectures as well as system board configuration for the multimedia embedded systems are very diverse. However, fortunately, as the general purpose embedded operating systems are getting widely used, the embedded applications are becoming built with standard development tools regardless of the underlying architecture. In addition, those development tools enable cross-developing environment, which consists of two computer systems with heterogeneous architecture; one is the target system and the other is separated high performance computer.

To fully utilize the effectiveness of the cross-development platforms and to support various hardware architectures, the target profilers should support various hardware architectures. The simplest way to achieve this is making them not depend on any hardware-specific features.

For example, the communication between clients and profilers has to be connected with standard network devices and protocols. The profiling action itself has to be accomplished only by software approach. As well, it is required that the client program can be run on various hardware architectures and operating systems.

In addition, to supporting the diversity of hardware architecture and operating systems, they should be able to monitor the behavior of target programs written in commonly used programming languages. The primitive embedded systems have evolved from dedicated logic circuits to programs written in C and assembly languages. However, due to the enhanced processor performance and the increased program complexity, the use of C++ grows to overcome the complexity and scalability. Thus the profilers should be able to analyze programs written in C++ as well as C or assembly..

E. Reporting Interface

Lots of existing open-source performance and behavior profilers only have text-based interfaces and generate only text-based result reports. Text-based products are adequate for analyzing applications which mainly consist of few repetitive-executed loops and functions. However, these are not suitable for showing information about dramatically many functions which are sequentially called only once for an operation that is the general characteristic of embedded system software.

Thus, as well as real-time graphical display of live data, the clients have to support generating reports in many standard document formats like **HTML** and **XML** to be viewed with existing browsers. Well-formatted and abstracted reports will reveal the problem points clearly. Finally, the client application should be able to save the raw data from profilers as a file. The raw data file will be transformed into other document formats later or used when users require fine-grained analysis of whole data to pinpoint problematic codes.

III. IMPLEMENTED PROFILERS

We implemented a prototype behavior profiler tool set for multimedia consumer electronics systems based on embedded Linux. It was designed to have the requirements defined in Section II. This section introduces the implemented tools and their principles.

A. Components

The implemented profiler tool set consists of four components; **Kernel Profiler**, **Application Profiler**, **System Call Profiler** and **Memory Leak Tracer**, respectively. The architecture of the profiler tool set is illustrated as shaded area in Figure 1.

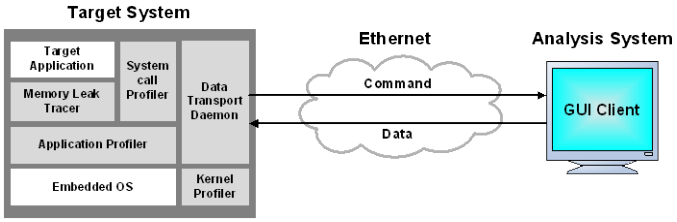


Fig. 1. The suggested profiler set is composed of 4 applications which collect different kinds of data, respectively. The gathered data are merged and sent to Analysis System by Data Transport Daemon.

All the suggested profilers have client-server structure. The clients are GUI programs which receive users' commands and show the corresponding results. Each client is implemented using Java to be used without recompiling in any platforms. Profilers, also servers themselves, are executed with targets concurrently in the same system. Each server gathers data fit for its purpose and forwards them to the corresponding client by network. The data forwarding is done by **Data Transport Daemon** which is an independent user-level task. To reduce the overhead transferring data is done with UDP instead of TCP.

Although each client has different purposes and functions, all clients have similar GUIs as shown in Figure 2. Commonly the GUIs have two buttons; one triggers start of profiling and the other stops profiling. As described in Section II, those two buttons enable the selective profiling of a certain interval the users want to profile.

Filtering unimportant functions and system calls is also possible. The clients send user-defined set to be filtered out to the profiling servers. Then the servers will ignore the data about the filtered units. This mechanism reduces the overhead from transferring unnecessary data.

The clients can draw call trees of functions in target applications or target kernels. The call trees are graphically displayed and they are able to be expanded and folded. Each entry of the call trees denotes a function in the target programs and it also shows how long each function took and how many times it was called. System Call Profiler also shows the information about the system call made in each function. The graphic information of the clients is to be updated on the fly as frequent as the update frequency, which is defined by users.

All clients use plain-text, HTML, XML and raw binary data

format for storing the data, while they show the real-time information graphically. The generated HTML or XML results are able to be displayed in any standard web browsers and the raw data can be read from our client to get the graphic information such as call trees at any time.

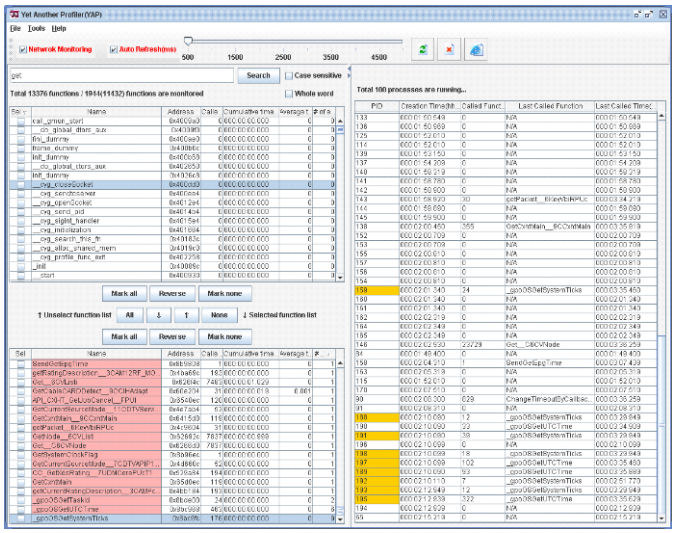


Fig. 2. Client GUI of application profiler consists of menu, control panel and data display area. Users can set function filters for the functions showed in the display area.

B. Profiling Methods

To grasp the execution time of each function call, recording the entry and exit time points of the function is necessary. This is enabled by placing an entry gate and an exit gate for each function.

An entry gate is a function called before the entry of the original function. On the contrary an exit function is called after the exit of the original function. Placing the entry and the exit gates is supported by many modern compilers. By implementing the entry and exit functions to log the current time when they are called, the execution time of the target function can be extracted from the log fie later.

Most of modern compilers provide an option, which wraps a function with the entry and exit gate functions, such as *-finstrument-function* compiling option in *gcc*, which is a popular open source compiler. Compiling with this option inserts the entry and exit gates to all the original functions as represented in Figure 3. Both of the entry and the exit gate functions get the caller function and the called function as their parameters. Thus, both the entry and exit gate functions know that who calls who at the moment of calling. This is used for counting the elapsed time caused by the target function along with the number of calling times.

In the detection of memory leak, knowing the parameters of the standard dynamic memory management functions such as *malloc* or *calloc* is essential. Placing the entry and exit gate for those functions is not an option here because the inserted functions know only the relationship between the caller and the callee, not the parameters the caller give to the callee.

Replacing the dynamic memory management functions with the profiling functions is used for this purpose. This can be done at loading time by *ld* loader. *ld* provides *wrap* option for this. With this option a specified function can be replaced by the other user-defined function. Naturally, the parameters given to the target function will be delivered to the replacing function.

In our implementation the dynamic memory management functions are replaced with a profiling function. The profiling function records the name of the original function and its arguments. After that, it calls the original function with the given parameters. This does not require recompiling while many of the existing memory leak detectors need recompiling [3].

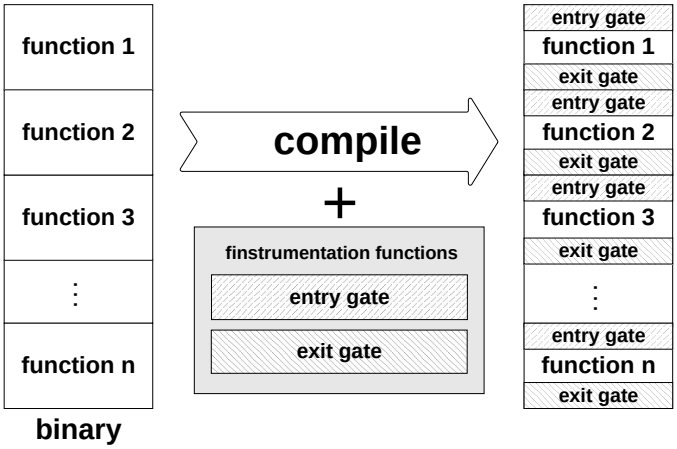


Fig. 3. Entry/exit gates are inserted to target functions with *–finstrument-functions* option at compiling

Finally, to identify the system call usage of a task, the task has to be monitored by the parent task with using *ptrace* system call. While a parent task uses *ptrace* to monitor its child tasks, the childs are suspended at every time the child use system calls and a signal is delivered to the parent. After suspending the child, the parent task can inspect the register and the memory of the child task by using *ptrace* again. System Call Profiler is a monitoring task which forks the target task and uses *ptrace* on it. By using the described method it checks who calls which system call at every time a system call is called by the target task.

C. Kernel Profiler

Kernel Profiler is developed to analyze the execution times of the functions inside the embedded Linux kernel. Since the kernel is located at the bottom of software hierarchy, to monitor the internal functions the profiler should be placed inside the kernel itself. Thus, the monitoring code of the suggested Kernel Profiler should be inserted into the kernel by kernel patching and recompiled before profiling.

The prototype Kernel Profiler was derived from *KFI* (Kernel Function Instrumentation) project [4]. *KFI* is an open-source kernel function profiler and also based on *-finstrument-functions* option of *gcc*. In other words, the target kernel must be compiled with the option before being monitored. The

prototype Kernel Profiler was modified from *KFI* to be suitable for the target environment.

We implemented a new client which can draw call graphs and generate reports in XML and HTML. The client was designed to be executed in the remote host. The profiling data generated by the inserted kernel functions are delivered to Data Transport Daemon with */proc* filesystem. Data Transport Daemon sends the data from */proc* filesystem to the remote client with network. Reversely, the filtering conditions and commands for suspending and resuming of profiling are sent to Data Transport Daemon from the remote client.

D. Application Profiler

The purpose of Application Profiler is to identify execution time and number of calls for each function in the target application on the fly. As described before, the prototype implementation uses the *-finstrument-functions* option. The profiling code for Application Profiler is provided as a library package. The target program is need to be recompiled with the library. After recompilation with the given library, entry and exit gate functions will be inserted the target binary.

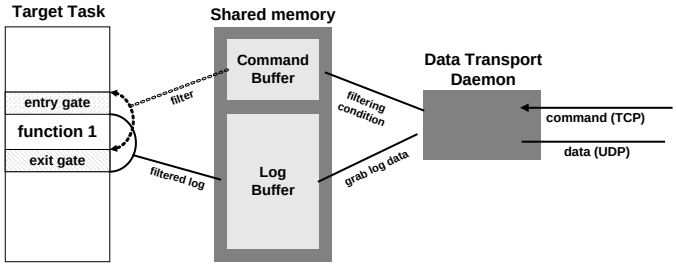


Fig. 4. Application profiler consists of data transport daemon, shared memory message box and profiling function library

Actual data transportation to the remote client is done by Data Transport Daemon. The communication between Data Transport Daemon and The profiling functions in the library is done with shared memory buffer. The sent data is removed from the buffer.

If the speed of generating data exceeds the speed of sending, then the profiler sends an error to the client. Overflow of the buffer occurs dependently on the network bandwidth and the characteristics of the target applications. In that case users may want to proceed the profiling with the focused set of functions. The filtering condition for the focused set is defined with GUI in the remote client. After the choosing the filtering conditions the remote client send them to Data Transport Daemon. Then Data Transport Daemon delivers it with the command buffer which is also shared memory space.

These selective function profiling hinders the construction of complete call trees since the filtered functions become missing links in call trees. Thus call trees are not displayed and only the calling times and the execution time of each function are showed in the filtered profiling mode.

E. System Call Profiler

As the general UNIX-style operating systems, Embedded Linux prohibits direct access to the hardware by user-level tasks for the security and the stability of the systems. Therefore the applications have to obey the rules; they control the peripheral hardware devices such as MPEG en/decoding chips and display brightness control only with system calls.

As a result, the meaningful information with regard to the hardware usage by an application is able to be extracted from the system usage history of that application. Especially, the round-trip time from user-level to kernel-level by a system call is a essential information for analyzing the delay from hardware operations. Also the calling point and calling times of system calls are important information. To get these information System Call Profiler runs a monitoring task prior to the start of the target application. The monitoring task forks the target application and traces it with *ptrace* system call.

The client of System Call Profiler is merged into the client of Application Profiler to show system call usages, behaviors of system call and functions as well as the call graph at the same time. Therefore System Call Profiler should be used with Application Profiler simultaneously. The client shows the elapsed time of each system call, the calling times of each system call and where a certain system call occurs.

F. Memory Leak Tracer

Most of existing memory leak detectors trace the dynamic memory operations from the start of the target application to the end of it [5]. If there left a dynamically allocated memory region which has not been freed until the end of the task, it is determined as memory leak. However, that method is unable to be applied to endlessly running tasks in consumer electronics embedded systems.

TABLE I
MONITORED DYNAMIC MEMORY OPERATIONS

Allocation	malloc, realloc, calloc, new, nothrow new
Deallocation	realloc, free, delete, nothrow delete

Decision of memory leak for an allocated dynamic memory region is done only by the intention of the developer. Since whether the allocated dynamic memory region will be used at future or not is unknown at a certain time, we cannot tell whether an existence of a certain allocated object is intentional or not.

Thus the prototype Memory Leak Tracer provides only the decision basis for memory leaks by showing the current status of the allocated dynamic memory objects.

Memory Leak Tracer intercept the calls for the known dynamic memory management functions listed in Table I by placing wrapping functions for them. The intercepting is done with the described *-wrap* option of GNU **ld**. The record about dynamic memory manipulation is transferred to the remote client right after the generation of it. Based on the received data the client shows the current allocated objects in each function, the illegally deallocated memory operations and the allocating operations overlapping existing objects.

The results display is updated with the newly received data on the fly. Thus users are able to find memory leaks with

inspecting the results in real-time. Also the features to make a snapshot file of the current status and save the results in the standard format such as XML and HTML will help the detection of memory leaks.

G. Simple Tuning Experience

The prototype tools were used for optimizing the embedded system of a commodity DTV(digital television). Although DTVs are rapidly spread, many customers complain about slow initializing and channel switching of them. Vendors try to resolve these phenomena and still suffer from those dissatisfactions.

The target system has a widely used general purpose embedded processor running at 250 Mhz, 128 MBytes of main memory and 32 MBytes of flash memory as its secondary storage. The operating system, system libraries and applications are stored in the flash memory. The temporary files will be located in a ram disk since there is no hard disk in the system. The source codes are written in C++.

The channel switching delay was discovered as about 1.8 s. and it is identified that the most of the time is consumed by system call handling which calls the device driver routines of the video decoder. The results are used as basis for improving performance of decoder chip.

Although the much portion of booting time was consumed by device initialization, significant amount of booting time was wasted by unnecessary initialization procedures. Table II shows the functions which are executed until the hardware initialization in Linux kernel and longer than 1000 us.

The profiled result shows that *cal_r4k* function consumes the longest time among the initializing procedures. *cal_r4k* function calibrates kernel variables about CPU clocks. This is done by looping an evaluative routine. However, this is unnecessary work since the hardware configuration hardly changes in CE embedded systems. Thus we replaced it with only assigning predefined values. After the modification *cal_r4k* was filtered out from the result because its execution time dropped down below 1000 us and the total booting time was reduced by 5 ms.

TABLE II
PROFILE RESULTS OF KERNEL BOOTING SEQUENCE

Function Name	Caller	Elapsed Time (us.)
start_kernel	init_arch	∞
setup_arch	start_kernel	1441
paging_init	setup_arch	1045
free_area_init	paging_init	1044
free_area_init_core	free_area_init	1044
time_init	start_kernel	5509
timer_setup	time_init	5509
cal_r4koff	timer_setup	5509

IV. RELATED WORK

Many performance profilers and memory leak detectors have been used for optimizing the performance of diverse applications and removing the erroneous dynamic memory management codes from them.

Processor vendors provide performance analyzer for their product. [6] is designed to provide diverse information about the binaries executed on some specific processors. It has powerful features and GUIs. The real-time remote analysis is also possible. However, it is not an open-source products and it heavily depends on the hardware specific profiling mechanisms of the vendor's processors.

Similarly, operating system vendors also provide performance analyzer for their operating systems. [7] exhibits integrated profiling environment from operating system kernel internal to the application for systems using Solaris 10 operating environment. However, it also depends on the operating system and remote profiling is not supported too.

gprof [8] is an open-source text-based profiler. An application which was compiled with a profiling option generates a file about the execution sequence and the execution time of the functions in the application source code. *gprof* analyzes and reports it with a human-readable text document. *gprof* is completely hardware-independent. However, the result is only possible to be made after the end of target applications and the generated data is stored in the local filesystem. Moreover it is designed to be used only for the user-level applications. Thus it is not adequate to be used in the target environment.

OProfile [9] is another open-source profile tool for monitoring the user-level tasks as well as the operating system internals. The profiling with *OProfile* does not require recompiling the target application since the actual performance monitoring is done by the kernel module provided by the *OProfile* package. The kernel module uses the performance monitoring registers which are given by *x86* processors for performance profiling. Thus it even produces instruction-level analysis. However, this approach is also hardware dependent. Therefore the calling number of each function is not possible to be identified and also drawing call graphs is only possible in the restricted hardware architectures.

Many kernel internal performance profiling tools have been introduced such as the KFI tool and *GILK* [10] which is a research product. A kernel is an inherently never-ending program. Thus the most kernel profilers support real-time profiling. However, profilers for Linux kernel are generally open-source products and thus they equips no GUIs or insufficient ones. Also most of them do not have the remote profiling feature.

Although most of the traditional memory leak detectors have weakness that they cannot show the status of dynamic memory objects in real-time [3], dynamic memory leak detectors are introduced for some specific environments such as garbage collected languages [5].

Another cutting-edge approach for memory leak detection is using a virtual machine to run a target application and audit in the virtual machine monitor level [11]. It emulates real-machine by just-in-time compiling of the target binaries and

thus the behavior of the target binaries are inspected at instruction level on the fly. As a result it can identify every dynamic manipulation of memory objects within the tasks and even show the processor cache activity during the run-time. However, the emulation slows down the execution speed severely. Considering the low performance of the target systems the emulation-based approach is inappropriate.

V. CONCLUSION

The target programs of the existing behavior-profiling tools have been mostly the computational applications which consist of few repetitively called functions that usually have long execution time. However, the embedded systems for multimedia consumer electronics have different characteristics. They consist of many short and sequentially executed functions. Moreover the underlying hardware architectures for consumer electronics systems are diverse and generally not enough to process large profiled data locally.

In fact, the performance optimization and debugging process are never less important in the consumer electronics embedded systems than the traditional computing systems. This paper analyzed the required properties for the behavior-profiler tool set which is suitable for developing and optimizing the multimedia consumer electronics embedded systems and suggested the desirable functionalities that the objective profilers should have. Based on them we designed and implemented a prototype profiler tool set for the multimedia gadgets. It consists of four profilers to cover from the inside of operating system kernel to memory leaks in user-level tasks.

As a case study, the prototype profiler was used for optimizing the reaction and booting time of a commodity digital television. We showed that it successfully identified the problematic points. With that experience we are assured that the specially designed profiler tools for the multimedia consumer electronics devices makes the optimizing process easier than the existing general purpose profilers for traditional computing systems.

REFERENCES

[1] S.-P. Moon, J.-W. Kim, K.-H. Bae, J.-C. Lee, and D.-W. Seo, "Embedded Linux implementation on a commercial digital TV system," *IEEE Transactions on. Consumer Electronics.*, vol. 49, no. 4, pp. 1402-1407, 2003.

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *American Federation of Information Processing Societies Conference Proceedings: Spring Joint Computing Conference*, vol. 30, pp. 483–485, 1967.

[3] D. Gay, R. Ennals, and E. Brewer, "Safe manual memory management," in *Proceedings of the 6th International Symposium on Memory Management*, pp. 2-14, 2007.

[4] N. M. Guire, "Kernel function instrumentation," white paper, OpenTech EDV-Research GmbH, 2005.

[5] M. Jump, and K. S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages,", in *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, pp. 31-38, 2007

[6] Intel Corporation, "Intel VTune Performance Analyzer How-To Guide," white paper, Intel Corporation, 2007.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of USENIX Annual Technical Conference*, 2004.

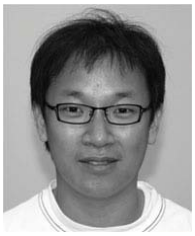
[8] S. L. Graham, P. B. Kessler, M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pp. 120-126, 1982.

[9] W. E. Cohen, "Tuning Programs with OProfile," *Wide Open Magazine*, Premiere Issue, pp. 53-62, bmind LLC, 2004.

[10] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder, "GILK: A dynamic instrumentation tool for the linux kernel," in *Computer Performance Evaluation / TOOLS*, pp. 220–226, 2002.

[11] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of PLDI*, 2007.

Seonyeong Park received the B.S. degree in computer science from Chungnam National University and the M.S. degree in computer science from Korea Advanced Institute of Science and Technology. Currently, she is a Ph.D. candidate in the Computer Science Division at KAIST. Her research interests include flash memory filesystems, embedded systems and power-aware computing.



. **Jinkyu Jeong**received the B.S. degree from the Computer Science Department, Yonsei University, and the MS degree in computer science from the Korea 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 real-time system, operating system, virtualization, and embedded system.

Heeseung Jo received the B.S. degree in computer science from Sogang University, Korea, in 2000, and worked as an engineer at the mobile platform service team of KTF, Korea, from 2001 to 2004. He received the M.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) and is a Ph.D. candidate of KAIST. His research interests include virtual machine, flash memory, storage system, and embedded system.

Joonwon Lee received the B.S. degree from Seoul National University in 1983 and the Ph.D. degree from the Georgia Institute of Technology in 1991. After working for IBM, he joined the faculty of the Korea Advanced Institute of Science and Technology (KAIST) in 1992. Currently, he is the faculty of the Sungkyunkwan University. His research interests include operating systems, low-power computing, and virtual machine.

Euiseong Seo received the BS, MS and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Korea. Currently, he is an assistant professor at Ulsan National Institute of Science and Technology. His research has been on power-aware resource management, real-time systems, and performance enhancement of virtualized systems.