PAPER

Reconfigurable Multi-Resolution Performance Profiling in Android Applications

Ying-Dar LIN[†], ***Member***, **Kuei-Chung CHANG**[††a)], **Yuan-Cheng LAI**[†††], ***and*** **Yu-Sheng LAI**[†], ***Nonmembers***

SUMMARY The computing of applications in embedded devices suffers tight constraints on computation and energy resources. Thus, it is important that applications running on these resource-constrained devices are aware of the energy constraint and are able to execute efficiently. The existing execution time and energy profiling tools could help developers to identify the bottlenecks of applications. However, the profiling tools need large space to store detailed profiling data at runtime, which is a hard demand upon embedded devices. In this article, a reconfigurable multi-resolution profiling (RMP) approach is proposed to handle this issue on embedded devices. It first instruments all profiling points into source code of the target application and framework. Developers can narrow down the causes of bottleneck by adjusting the profiling scope using the configuration tool step by step without recompiling the profiled targets. RMP has been implemented as an open source tool on Android systems. Experiment results show that the required log space using RMP for a web browser application is 25 times smaller than that of Android debug class, and the profiling error rate of execution time is proven 24 times lower than that of debug class. Besides, the CPU and memory overheads of RMP are only 5% and 6.53% for the browsing scenario, respectively.

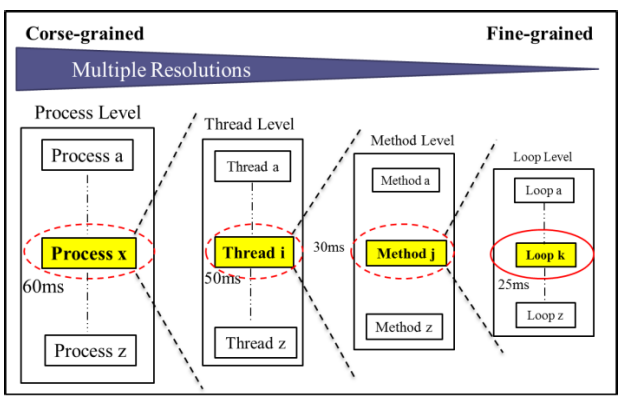
key words: *time profiling, multi-resolution profiling, android, reconfigurable profiling*

1. Introduction

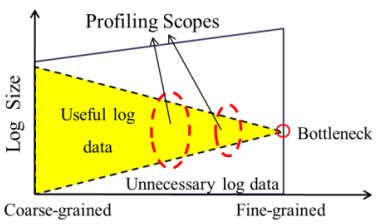
In designing efficient embedded applications, two key design issues should be considered. First, the execution time of embedded applications should be optimized because they have to run on embedded devices with limited computing capability. Second, the energy consumption should be minimized because the battery power is a crucial resource for embedded devices. Therefore, developers need to identify the hotspots in the program and optimize their applications according to the analyzed results.

There are many existing profiling tools which can help developers to identify bottlenecks, such as PowerScope [1] and Gprof [2]. Some of them can only profile applications at a single resolution. For example, the PowerScope can only provide a coarse-grained profiling resolution (e.g. process level) to analyze the energy consumption of a process. Thus, developers using PowerScope have the difficulty to identify precise bottlenecks within a process. On the other hand, Gprof provides a fine-grained profiling resolution (e.g. function level), and thus the developers can analyze the time information of a process in more details. However, they need to spend a lot of time to analyze logs of the detailed profiling information to identify performance bottlenecks.

In this article, we propose a new multi-resolution profiling scheme, named reconfigurable multi-resolution profiling (RMP), to help developers to profile the embedded applications flexibly. It can profile the execution time at various profiling resolutions, such as process level, thread level, function level (method level), and loop level. RMP can efficiently profile execution time for different profiling scopes with limited log space. The profiling scope can be defined and controlled by user configurations, such as profiling resolution and user-specific filtering rules. Figure 1 gives an example of profiling; the profiling can be started at any specific coarse-grained resolution (e.g. process level). Developers can zoom into next fine-grained resolution (e.g. thread level) when the hotspot of the application at the coarse-grained



(a) Multi-resolution profiling



(b). Gradual Profiling Phases

Fig. 1 The concepts of RMP.

Manuscript received June 25, 2012.
Manuscript revised March 17, 2013.

[†]The authors are with the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan.
[††]The author is with the Department of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan.
[†††]The author is with the Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan.
a) E-mail: changkc@fcu.edu.tw (Corresponding author)

DOI: 10.1587/transinf.E96.D.2039

Table 1 Comparisons of profiling tools.

	Tool Name	Profiling Method	Profiling Resolution	Features
Single-resolution profiling Tools	Gprof	Instrumentation	Function Level	• Instrumented by compiler • Interleaved logs problem
	KFT	Instrumentation	Function Level	• Instrumented by compiler • Provide complete call graph
Multi-resolution profiling Tools	LTTng	Instrumentation	Process – statement	• Low overhead • Not support java • Friendly GUI
	Debug class	Instrumentation	Process – method	• Android built-in tool • Friendly GUI – Traceview
	Oprofile	Sampling	Process – statement	• Supported by Android • Not support java
	HPCToolkit	Sampling	Process – loop	• Hardware dependency • Friendly GUI
	Intel VTune	Sampling	Thread – statement	• Hardware dependency • Friendly GUI

resolution is identified, such as process x in Fig. 1 (a). When the hotspot of process x is identified as thread i, the same zoom-in process can be used again. By RMP, the exact bottleneck can be identified with small log space because it only stores and analyzes the necessary profiling logs at a specific scope during profiling, as shown in Fig. 1 (b).

For measuring the execution time in RMP, we use the similar approach proposed by LTTng [3] which instruments some probes in the source code to be activated at runtime to record execution information about a program. The major difference between LTTng and RMP is that the probes in RMP can be configured to change the profiling scope. The RMP has been mounted onto SourceForge website as an open source tool (http://sourceforge.net/p/rmptool/).

The remainder of this article is organized as follows. Section 2 discusses related work of performance profiling. We describe the architecture of RMP for Android system in Sect. 3. Section 4 presents the experimental environment and discusses evaluation results. Conclusions and future work are given in Sect. 5.

2. Related Work

Existing time profiling techniques can be divided into two main categories: instrumentation and sampling. Instrument-based profiling tools instrument some profiling points into a program, and log events will be recorded when these instrumentation points are triggered. Gprof [2] and Kernel Function Trace (KFT) use the compiler-assisted capabilities to automatically instrument profiling points at entry and exit of every function. However, these tools can only provide the function-level profiling results because they just instrument profiling points at the entry and exit of functions.

Linux Trace Toolkit Next Generation (LTTng) [3] provides a programming interface to instrument the source code. The instrumentation points are managed with probes and every probe can be configured to be "on" or "off" state at runtime.

Debug class, an Android built-in java class, provides a way to create log and trace the execution of an Android application. The application can be profiled without any specific instrumented profiling code. However, the collected profiling information will be very huge and complex. The users can use debug class to specific the profiling range for gathering the interested profiling results. After profiling, TraceView can analyze the log and show the execution information from process level to method level.

Sampling-based profiling tools utilize hardware performance counters, embedded in most modern CPUs, to record program execution information, such as program counter (PC) and cache misses. Then the results can be correlated with the structure of source code. Representative tools include Oprofile, HPCToolkit [5], and Intel VTune. Most of them can analyze the fine-grained profiling traces and show results from a coarse-grained resolution to a fine-grained resolution of details by a GUI tool or a formatted text.

Existing energy profiling can be divided into three categories: simulation approach, measurement approach, and estimation approach. Simulation approaches create virtual hardware platforms to simulate energy consumption behavior for energy profiling [6], [7]. Measurement approaches measure energy consumption with digital power meters directly [1], [8]. The power meter is connected to a platform which uses the time-driven sampling approach to periodically trigger the power meter to record the energy consumption. Estimation approach counts the requests of hardware components for each process [9], [10]. The amount of requests can be translated into energy consumption using the energy estimation model which includes the estimation formula and the power table.

Battery Use has been embedded in Android systems from version 1.6. It can provide the information of energy consumption for each process. An enhancement of Battery Use [4] uses a two-phase calibrating approach to create new estimation formulas and a more correct power table. It improves the accuracy of estimating energy consumption with the error rate below 10%.

Table 1 summarizes the above discussed profiling tools. Gprof, Kernel Function Trace (KFT), and all energy profiling tools are based on single-resolution profiling technique, and others support multi-resolution profiling to help users easily analyze the bottlenecks by GUI. However, all multi-resolution profiling tools normally log all the fine-grained profiling traces at runtime and show multi-resolution profil-

ing results on GUI at the post-analysis phase. However, the large amount of logging information cannot be accommodated by the resource-constrained embedded systems, such as Android. In this article, we propose an approach to solve this limitation.

3. Reconfigurable Multi-Resolution Profiling for Android

This section first describes the architecture and the detailed methodology of the reconfigurable multi-resolution profiling approach (RMP). Afterwards, the profiling flow of RMP is presented.

3.1 RMP Architecture

Figure 2 depicts the architecture of RMP which consists of five components: instrumentation component, control and display component, time profiling component, and logs collection and correlation component. For changing profiling scope without recompiling, the instrumentation component is designed to analyze and instrument all necessary profiling points in the source code. The control and display component displays the profiling point configurations and the profiling results for developers. The time profiling component can record the time information of profiling scope based on the configurations of the control and display component. Finally, the logs collection and correlation component can store time logs and correlate them at fine-grained resolutions.

3.2 Instrumentation Component

In RMP, the instrumentation component instruments profiling points to the whole necessary locations of the application and the framework at first. However, only parts of the framework API will be called by the profiled application; if all profiling points are instrumented to the whole framework, the useless profiling points may cause extra unnecessary overhead for other applications. Therefore, the instrumentation range of the application should be analyzed by the instrumentation range analyzer before the profiling points are instrumented in the framework. The all methods of the profiled application and the framework methods called by the application are defined as the *instrumentation range*.

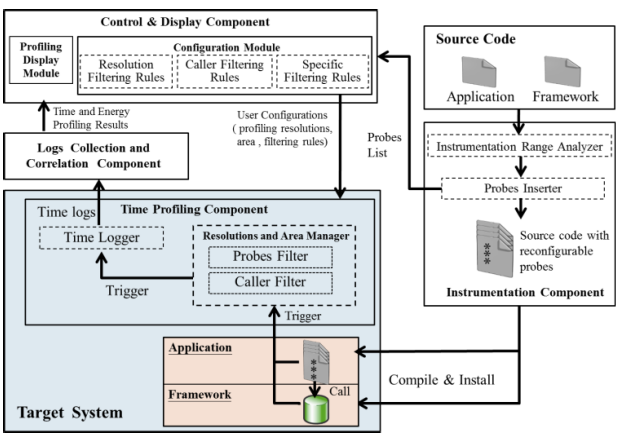


Fig. 2 Profiling architecture of RMP.

When the instrumentation range is identified, the probes inserter can start to instrument *profiling points* into the application and framework source code. The pair of profiling points of a method or a loop map to a *probe* which is a control unit using by developers to control the profiling scope. Each probe is recorded in a probe list with its probe name, resolution definition, and on/off status. The profiling resolution also can be recognized by specific functionality of the method (ex. the run() method in Java can be regarded as thread level). After instrumentation, the application and framework source code can be compiled and installed on the target system to do multi-resolution profiling.

There are four profiling resolutions for Android applications in this implementation. First, *process level* includes all profiling results of components in an application because the application components are essential building blocks of an Android application. Therefore, the process level is used to analyze the bottlenecks of components in an application. The execution period of Android *activity* is from onCreate() state to onDestroy() state, but the activity may be switched to other activities between onCreate() state and onDestroy() state when users change to other UIs of the application. Therefore, the actual execution time of an activity can be estimated by the periods of onCreate(), onDestroy(), and each part of the visible phase. The execution time of a Android *service* can be estimated by the period from onCreate() state to onDestroy() state, and the execution time of a Android *broadcast receiver* can be estimated by the time of onReceive(). However, the Android *content provider* doesn't have the independent execution time because it is always called by other components in general. Therefore, the execution time of the content provider has been included in other components.

The second resolution is *thread level* which includes the profiling results of run() method implemented in a thread class. This resolution is used to analyze the bottlenecks of threads in a process. The third resolution is the *method level* which includes the profiling results of each normal method. This resolution is used to analyze the bottlenecks of methods in a thread. The last resolution is the *loop level* which includes the profiling results of each loop in a method. This resolution is used to analyze the bottlenecks of loops in the method.

3.3 Control and Display Component

The control and display component gets the probe list from the instrumentation component and shows the probe control information for users with GUI. The user can turn on or turn off the probes to control the profiling scope by the *configuration module*. In addition, the *profiling display module*

shows the energy and time profiling results of the application. The profiling scope can be identified by the configurations of the profiling resolution and the profiling area. The *profiling resolution* is defined by the language-based level, such as process, thread, function (method) and loop. The developer can zoom a coarse-grained resolution into a fine-grained resolution to analyze bottlenecks by the resolution filter, and the filter will turn the probes on or off according to the user resolution configurations. The *profiling area* is defined by the set of all active probes which will be profiled. For example, when the profiling is working at the method level of thread *i*, the profiling area includes all method-level probes in thread *i*. Also when the profiling resolution zooms into the method *k* of thread *i*, the profiling area includes all loop-level probes in method *k*.

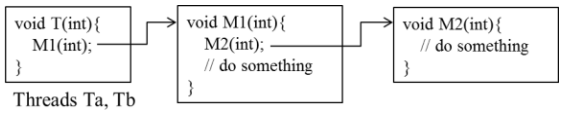
3.4 Time Profiling Components

The *time profiling component* checks the probes status (on/off) and manages the caller list based on the user configurations before it records the time data. When the application executes on the target system, the probes will be triggered and send log events to the time profiling component. If the status of a probe is off, the events related to the probe will be skipped; if the status of a probe is on and the caller of the probe has been recorded in the caller filter, the log events will be recorded into log space. The size of the log space can be set by developers. When the log space of time profiling component is out of bound or the profiling is finished, the logs will be removed from memory to other large storages.

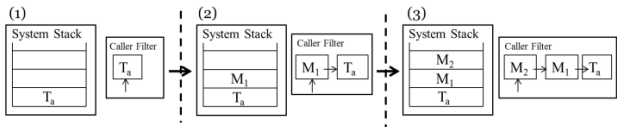
A method may be executed by either profiled threads or other non-profiled threads. Therefore, a *caller filter* is designed in the time profiling component to solve this problem. The user can set some root processes or methods as interested profiling targets in the caller filter, and then only the methods that are called by the methods in the caller filter can be profiled. The callers of the methods recorded in the caller filer will also be dynamically recorded into the caller filter at runtime to identify all methods needed to be profiled.

Figure 3 shows an example of the caller filtering. We assume methods 'M1' and 'M2' are called by threads 'Ta' and 'Tb', as shown in Fig. 3 (a). 'Ta' is the interested profiling thread and is stored in the caller filter before execution shown as Fig. 3 (b1). In Fig. 3 (b2), 'M1' will be recorded into the caller filter when it is called by 'Ta' because the caller of 'M1' is 'Ta' which has been recorded in the filter. 'M2' will be recorded into the caller filter as well as 'M1' shown as Fig. 3 (b3). When the method is finished, it will be removed from the filter. By this way, the methods will be correctly profiled according to the caller filtering. Figure 3 (c1) shows a counterexample; 'Tb' is not our interested profiling thread and it will not be recorded into the caller filter when it is executed. Thus methods 'M_1' and 'M_2' will not be recorded into the caller filter and not be profiled too, as shown in Fig. 3 (c2) and Fig. 3 (c3).

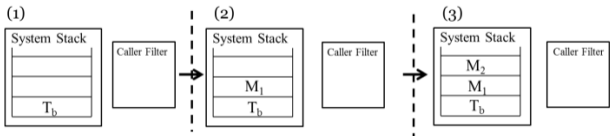
In addition, the caller filter also controls profiling area



(a) The pseudo code of a thread and the invoked methods.



(b) Ta is the interested profiling thread.



(c) Tb is the uninterested profiling thread.

Fig. 3 Examples of the caller filtering.

when zooming in other resolutions. For example, when the developer zooms into the loop level in method *k*, the profiling area can be limited in the method *k* by the caller filter. It can avoid recording loop-level events of other methods. Also the setting of specific filtering rules according to user preference can help the developer to narrow down profiling area during the step-by-step profiling process, such as execution time over than one sec and executing times over than five. These specific filtering rules can also help developers to exclude uninterested log events and to further reduce the log size.

When the profiling points in the application and framework are triggered, the time profiling component receives log events, checks probes status, and manages caller filter based on the user configurations before it logs the time data into the log memory. These actions need to be implemented efficiently because they may influence the accuracy of the profiled time results and the profiling performance if they spend too much time.

The time profiling component was written in C language which has higher performance than java language. We use *java native interface (JNI)*, an efficient approach to call C program from Java program on Android [11], for profiling points to call the time profiling component. The time profiling component is implemented as JNI native library which will be loaded when the application starts to execute.

Figure 4 shows an example of calling the time profiling component using JNI. The profiling point calls the time profiling component through JNI and sends its probe name, caller name, and attribute for checking. The API, Thread.currentThread.getStackTrace(), is used to collect the stack information and select out the caller name. However, it spends too much execution time. Since only the caller name in the stack is required, we implemented a new API, Thead.currentThread.getCaller(), to only returns the caller

```
...
String caller_name = Thread.currentThread ().getCaller();
probe ("com.android.browser.TabControl.restoreState ", caller_name , true);
...
```

JNI call

```
void nativeProbe (JNIEnv *env, jobject obj, jstring jname, jstring jcallername, jboolean point) {
  const char *nname = (*env)->GetStringUTFChars (env, jname, 0);
  const char *ncallername = (*env)->GetStringUTFChars (env, jcallername, 0);

  if(point == JNI_TRUE) {
    if(isActiveProbe (nname) == 1)
      if(isCaller (ncallername , nname ) == 1)
        activeLog (nname );
  } else {
    if(isActiveProbe (nname) == 1)
      if(removeCaller (nname ) == 1)
        activeLog (nname );
  }

  (*env )->ReleaseStringUTFChars (env, jname , nname );
  (*env )->ReleaseStringUTFChars (env, jcname , ncname );
}
```

Fig. 4 Example of calling the time profiling component.

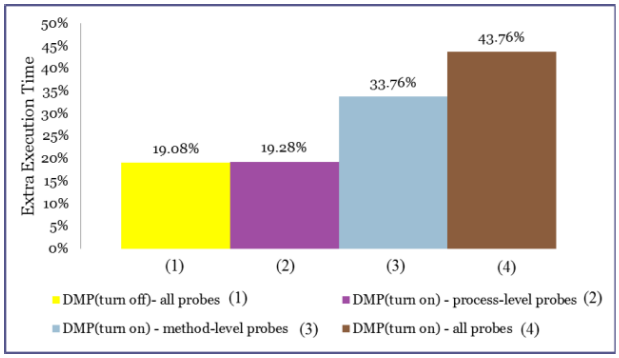


Fig. 5 The overhead of execution time for RMP.

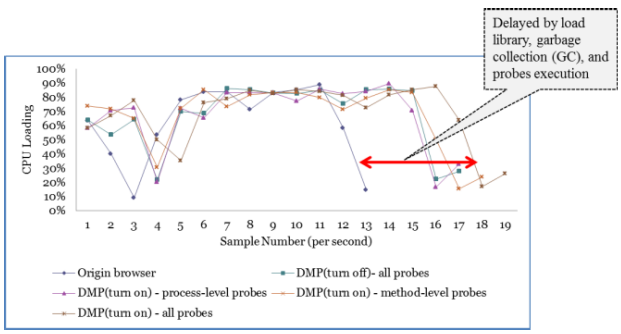


Fig. 6 CPU loading during profiling.

name to save execution time.

3.5 Logs Collection and Correlation Component

The time profiling component generates performance results to users. The logs can be stored for off-line analysis to get more detailed profiling results. In addition, we can try to correlate the process-level energy consumption with fine-grained time profiling results to get the fine-grained energy consumption using Battery Use [4] energy information in the future.

4. Evaluation Studies

4.1 Evaluation Environment and Scenarios

The version of Android platform is Android open source project (ASOP) 1.6. Most evaluation experiments are based on the *web browsing* because the Android default browser spends a lot of time when it loads and shows web pages. The browsing scenario starts from the user touching the screen to start the browser until the default page have been loaded and shown on the screen.

4.2 Experimental Results

4.2.1 Execution and Memory Overheads

Figure 5 shows that the execution time of the browser is extended 19.08% when all probes are turned off (not record any log events); and the execution time is extended 43.76% when all probes are turned on (record all log events). The extended time is composed of three parts. First, the time profiling component needs to spend time to check probe on/off status, to filter caller, and to store time results for each probe. Second, the VM spends time to initiate the time profiling component (JNI shared library) when it loads and initiates classes which have been instrumented with probes. Third, the time profiling component stores time results in the memory and removes it from the log memory to other large storages when the log space is full. Therefore, the time profiling component will allocate and free memory several times. These actions will trigger the VM to do garbage collections.

Figure 6 depicts the CPU loading of the browsing scenario. CPU loading variations of all situations are similar because the computation of the time profiling component does not take much CPU resource. The average extra CPU loading is 5% when all probes are turned on. Although the execution time of the instrumented browser extended ranging from 19.08% up to 43.76% shown in Fig. 5, it does not cause too much CPU loading because the additional profiling tasks do not take too much computation resource.

Figure 7 depicts the log memory usages during profiling. We use *libpagemap* function to periodically measure the memory usage of the corresponding linux-level process (com.android.browser) for the profiled Android application in Linux level. We sample the memory usage of the process every two seconds. The used memory of the instrumented browser is more than that of the origin browser in general because the time profiling component stores the time results in the memory. The extra overhead of log memory can be controlled below 6.53% when all probes are turned on. This is because RMP can change profiling scope to save the log space. The used memory of the instrumented browser is sometimes less than that of the origin browser during the profiling because the actions of allocating and releasing memory triggers the VM to do garbage collections (GC). Figure 7 shows the situation during sample 3 and sam-

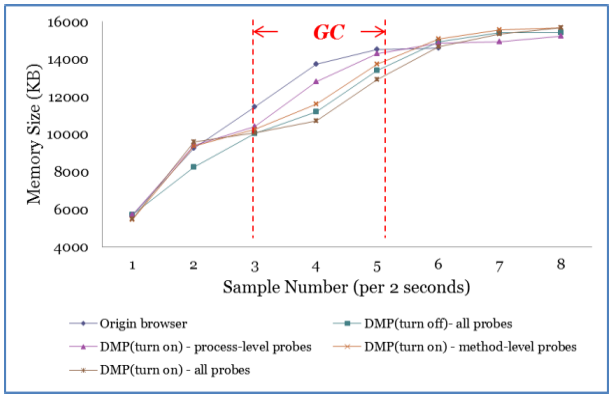


Fig. 7 The overhead of memory usage.

ple 5.

4.2.2 Time Profiling Accuracy

Methods getCaller() and Probe() are instrumented in each method to get the caller name and trigger the time profiling component to record time data, respectively. Both of them take extra execution time during profiling that will be included in the time results of RMP; therefore, the extra time needs to be deducted for accuracy of time profiling results.

We select a method which has been instrumented with probe statements, and it invokes a child method which has been instrumented too. Another case is that the instrumentation statements are removed from the child method. The extra time of a probe can be calculated by the difference between these two test cases, and the measured execution time of a probe is 0.488 ms. Therefore, this consumed time of probes is deducted from the profiled time results of RMP according to the number of probes in methods. The time 0.488 ms is estimated for an application probe, but the extra execution time of a framework probe is different. The framework probe has one more stage in the time profiling component to check whether the caller is from the target profiling application. Therefore, the extra execution time of a framework probe is 0.631 ms got from the similar experiment for framework probes.

In Table 2, some methods with stabler execution time are selected to evaluate the error rate of time profiling results. The reference time is got from the API *System.nanotime()*. This API is instrumented into the entry point and exit point of these methods to get the execution time in nanosecond. According to the above mention, the time results of RMP minus the extra execution time of probes are as the final profiling results. The table shows that the error rate of RMP is below 10.21%. On the contrary, the error rate of debug class is over than 43.17% because the execution time of a method is estimated by the period between the entry of the method and the entry of next executed method in debug class. When the time results include the execution time of calling the next method, the results could be interfered with other miscellaneous factors. From Table 2, the average profiling error rate of execution time of

Table 2 Error rate comparisons of time profiling.

	onPause	addObserver	resetLockcon	buildTitleUrl
System.nanotime( ) (ms)	14.518	1.221	1.312	1.068
RMP (ms)	15.562	1.167	1.282	1.177
Error rate (%)	**7.19**	**0.04**	**2.29**	**10.21**
Debug Class(ms)	8.25	2.417	2.099	4.061
Error rate (%)	**43.17**	**97.95**	**59.98**	**280.24**

Table 3 Functional comparisons with debug class.

	Profiling resolution	Profiling target	Log size	Integral profiling problem
Debug class + Traceview	Process - Method	Application Framework Core library of VM	Large	Yes
RMP	Process - Loop	Application Framework	Small	No

Table 4 Overhead comparisons with debug class.

	Extra execution time	CPU loading	Used memory
Debug class	39.72%	4%	76.3%
RMP	19.08~43.76%	5%	6.53%

RMP is 24 times lower than that of debug class.

4.2.3 Overhead Comparisons with "Debug Class + Traceview"

The *debug class* with *Traceview* is a default time profiling tool working on most Android product devices, and thus selected to be compared with RMP. The debug class collects all time results of methods and stores the results in the memory at runtime. It will transfer time results from memory to the SD card after the profiling is finished. Therefore, the debug class requires large memory space to store the time results. Traceview only provides the round times and total execution time for a method. Also the execution time of each round for a method is the average result. However, a method may spend different execution time in different task by parameters. Therefore, Traceview cannot help us to analyze the execution time of each round for a method, named integral profiling problem. Table 3 shows the comparisons between debug class + Traceview and RMP.

Table 4 shows the overhead comparisons of debug class and RMP. The debug class extends 39.72% overhead of execution time, 4% CPU loading, and 76.3% memory overhead. The overhead of memory is very large because debug class collects time results of all methods and stores them in the memory. The execution time overhead and CPU loading of debug class and RMP are close. However, the memory overhead of RMP is much smaller than that of debug class because RMP is a reconfigurable multi-resolution profiling solution which can save the log space. The total log size of the debug class is 2.99 MB for this experiment. The

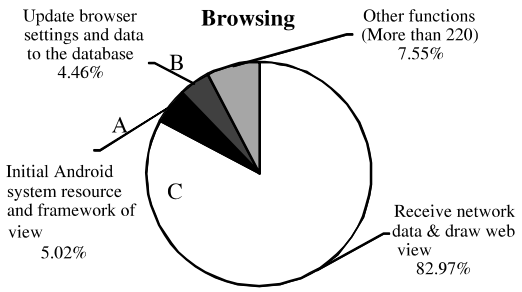


Fig. 8 The execution time distributions of the browsing scenario.

same experiment using RMP takes eight rounds to identify the same bottleneck, and the average log size of each round is 15 KB. It is easily observed that the required memory space using RMP for a web browser application is 25 times (2.99 MB/15 KB x 8) smaller than that of Android debug class.

Figure 8 shows the detailed information about the profiled execution time for the browsing scenario. It was modeled as a pie chart according to the module's functionality. The "A" part takes 5.02% of the browsing time for initiating the Android resource (Ex. SGL) and the framework of view. The "B" part takes 4.46% of the browsing time for updating browser settings (Ex. plugins status) and data to the database. The "C" part takes 82.97% of the browsing time for receiving network data and drawing web view. The actions of receiving network data and drawing web view are implemented in C/C++ library, so we cannot know the detailed time distributions for these actions. But the bottleneck is at the web view drawing by analyzing the library source code manually. The view of 3D is drawn by OpenGL which uses the graphic processing unit (GPU) for hardware acceleration, but the view of 2D is drawn by SGL without any hardware acceleration. It is slow when the web page is drawn by the embedded CPU. Therefore, if SGL can support hardware acceleration of 2D drawing, the performance of the embedded web browser can be raised.

For the browsing application, the size of the original Android framework source code is 149.8 MB, and the size of the compiled image file is 55.49 MB. The size of the instrumented framework source code is 150.2 MB, and the size of the compiled image file is 55.56 MB. The overhead of the instrumented framework source code is 0.27%, and the overhead of the instrumented image file is 0.13%.

4.3 Implementation Issues

In order to implement the proposed RMP on Android, the application and the framework source codes have to be preprocessed for profiling. RMP has to analyze the instrumentation range to identify what framework methods are called by the application. Then, we can instrument all necessary profiling points into the application and the framework at once. When all probes are inserted into the framework and the application, the whole framework and application have to be recompiled and restarted.

We also added some tools to assist the profiling process. First, we use *Jindent*, a source code formatter tool, to unify the source code format of the application and the framework. The unification format of the source code can minimize the complexity of the instrumentation algorithm. Second, we use *ctags* to get the location of the application methods and the framework methods. The results of ctags include the name of method, file name of the method, and line number of the method in the file. We implement an automatic script to scan out what framework methods are called by the application according to the results of ctags. Then, users can run the script in different operating systems directly. The automatic script is written in the *python* language because the python language is easy and it can work on most popular operating systems, such as Windows and Linux. Finally, we implement another automatic script to instrument entry and exit profiling points into the application and the framework methods with the resolution definition.

5. Conclusion and Future Work

This work proposes the reconfigurable multi-resolution profiling approach to profile execution time of applications with limited log space to identify the bottlenecks of applications on the resource-constrained embedded system. We have shared the implemented RMP tool on the SourceForge website. The evaluation studies have proven that our implementation has minor CPU loading overhead (5%) and memory overhead (below 6.53%). Although the execution time of the instrumented application is increased ranging from 19.08% to 43.76% depending on the number of active probes, the overhead can be correctly deduced from the profiled execution time to recover accurate profiling results. The accuracy of execution time results is evaluated with the average error rates below 10.21%, which is 24 times lower than that of debug class. In the future, we shall investigate energy consumption with the proposed RMP tool.

Acknowledgments

This work was supported in part by the NSC under Grant No. NSC 101-2219-E-035-001, NSC 101-2219-E-011-003 and NSC 101-2219-E-009-013.

References

[1] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," 2nd IEEE Workshop on Mobile Comp. Syst. and Apps, pp.2–10, Feb. 1999.
[2] S.L. Graham, P.B. Kessler, and M.K. Mckusick, "gprof: A call graph execution profiler," SIGPLAN Notices, vol.17, no.6, pp.120–126, June 1982.
[3] M. Desnoyers and M. Dagenais, "LTTng: Tracing across execution layers, from the Hypervisor to user-space," Proc. Ottawa Linux Symposium, pp.101–105, July 2008.
[4] Y.-C. Yu, Calibrating parameters and formulas for process-level energy consumption profiling, Master Thesis, National Chiao Tung University, Taiwan, June 2010.
[5] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J.

Mellor-Crummey, and N.R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," Concurrency and Computation: Practice and Experience, vol.22, pp.685–701, 2010.

[6] T.L. Cignetti, K. Komarov, and C.S. Ellis, "Energy estimation tools for the palm," Proc. 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pp.96–103, 2000.

[7] S. Gurumurthi, A. Sivasubramaniam, M.J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L.K. John, "Using complete machine simulation for software power estimation: The softWatt approach," Proc. 8th International Symposium on High-Performance Computer Architecture, pp.141–150, 2002.

[8] W. Choi, H. Kim, W. Song, J. Song, and J. Kim, "ePRO-MP: A tool for profiling and optimizing energy and performance of mobile multiprocessor applications," Scientific Programming - Software Development for Multi-core Computing Systems, vol.17, no.4, pp.285–294, Dec. 2009.

[9] T. Do, S. Rawshdeh, and W. Shi, "pTop: A process-level power profiling tool," Proc. 2nd Workshop on Power Aware Computing and Systems, 2009.

[10] K.S. Banerjee and E. Agu, "PowerSpy: Fine-grained software energy profiling for mobile devices," Proc. Int. Conf. Wirel. Netw., Commun. and Mobile Comput., vol.2, pp.1136–1141, June 2005.

[11] L. Batyuk, A.-D. Schmidt, H.-G. Schmidt, A. Camtepe, and S. Albayrak, "Developing and benchmarking native linux applications on android," MobileWireless Middleware, Operating Systems, and Applications, pp.381–392, 2009.

Ying-Dar Lin is Professor of Computer Science at National Chiao Tung University (NCTU) in Taiwan and also an IEEE Fellow. He received his Ph.D. in Computer Science from UCLA in 1993. He served as the CEO of Telecom Technology Center during 2010–2011 and a visiting scholar at Cisco Systems in San Jose during 2007–2008. Since 2002, he has been the founder and director of Network Benchmarking Lab (NBL, www.nbl.org.tw), which reviews network products with real traffic. He also co-founded L7 Networks Inc. in 2002, which was later acquired by D-Link Corp. He recently, in May 2011, founded Embedded Benchmarking Lab (www.ebl.org.tw) to extend into the review of handheld devices. His research interests include design, analysis, implementation, and benchmarking of network protocols and algorithms, quality of services, network security, deep packet inspection, P2P networking, and embedded hardware/software co-design. His work on "multi-hop cellular" has been cited over 600 times. He is currently on the editorial boards of IEEE Transactions on Computers, IEEE Computer, IEEE Network, IEEE Communications Magazine - Network Testing Series, IEEE Communications Surveys and Tutorials, IEEE Communications Letters, Computer Communications, Computer Networks, and IEICE Transaction on Information and Systems. He recently published a textbook "Computer Networks: An Open Source Approach" (www.mhhe.com/lin), with Ren-Hung Hwang and Fred Baker (McGraw-Hill, 2011). It is the first text that interleaves open source implementation examples with protocol design descriptions to bridge the gap between design and implementation.

Kuei-Chung Chang received the Ph.D. degree in computer science from National Chung Cheng University in 2008. He is Associate Professor of Department of Information Engineering and Computer Science at Feng Chia University in Taiwan. His research interests include system-on-chip, network-on-chip, embedded system, and multi-core system.

Yuan-Cheng Lai received the Ph.D. degree in computer science from National Chiao Tung University in 1997. He joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology in 2001 and has been a professor since 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking. He can be reached at laiyc@cs.ntust.edu.tw.

Yu-Sheng Lai received the M.S. degree in computer science from National Chiao Tung University in 2011. He is now a project engineer at the R&D division of software business unit of Acer Inc. His research interests include the development of bootloader and embedded operating systems, and performance evaluation of embedded software.