國立交通大學

資訊科學與工程研究所

碩士論文

調整解析度於剖析Android應用程式的耗時與耗電

Reconfiguring Resolutions in Profiling Time and Energy on Android Applications

研 究 生：賴育聖

指導教授：林盈達　教授

中華民國一百年六月

調整解析度於剖析 Android 應用程式的耗時與耗電

Reconfiguring Resolutions in Profiling Time and Energy on Android Applications

研究生：賴育聖　　Student：Yu-Sheng Lai

指導教授：林盈達　　Advisor：Dr. Ying-Dar Lin

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2011

Hsinchu, Taiwan

中華民國一百年六月

調整解析度於剖析 Android 應用程式的耗時與耗電

學生：賴育聖　　　　　　　　　　　　　指導教授：林盈達

國立交通大學資訊科學與工程研究所

摘要

嵌入式裝置上的應用程式於運算時經常受到運算資源和能源上的嚴格限制。然而因資源受限而導致運算時間過長及消耗大量能源是不被使用者接受的，因此應用程式是否能有效利用嵌入式裝置上的資源是一個重要的研究議題。現有的執行時間和耗能分析工具可幫助開發者找出應用程式其效能與耗能的瓶頸，但當這些工具提供詳細資訊給使用者的同時需要大量的記憶體空間來儲存分析過程中所產生的資訊，而記憶體空間在嵌入式裝置上也經常受到嚴格的限制。這篇論文提出一個於移動式裝置產品上可變更設定以切換解析度來多層級剖析耗時與耗電的工具。此工具先插入分析程式碼於所有目標應用程式的原始碼，然後再藉由設定所插入的分析程式碼來切換分析的範圍並過濾掉大量不需分析的資訊以達到減少分析資訊量的目的。分析出的資訊量在瀏覽網頁的情境下比 Android 的 debug class 少 25 倍，且我們工具分析的時間資訊在同樣的瀏覽網頁情境下其數據誤差率比 debug class 低 24 倍，這是因為 debug class 的數據是藉由 java methods 的進入時間點來計算而造成數據並不精確。另外此工具對於 CPU 和 memory 所造成的額外負擔在瀏覽網頁的情境下分別為 5%和 6.53%。最後，經由我們設計的工具剖析後發現瀏覽網頁的效能瓶頸在於 2D 繪圖函式是使用 CPU 來繪畫網頁內容，未使用任何硬體加速導致用繪畫速度很慢。

關鍵字： 耗時分析，耗能分析，多層級分析，Android

Reconfiguring Resolutions in Profiling Time and Energy on Android Applications

Student: Yu-Sheng Lai Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

Abstract

The computing of applications in embedded devices suffers tight constraints on computation and energy resources. Thus, applications spending long execution time and large energy consumption of embedded applications are not acceptable by users. The existing execution time and energy profiling tools can help developers to identify the bottlenecks of applications. However, the profiling tools need large memory space to store detailed profiling results at run time, causing that they are infeasible on embedded devices. In this thesis, a reconfigurable multi-resolution profiling (RMP) approach is proposed to handle the issue on off-the-shelf product devices. It instruments all profiling points into source code of targeted applications and configures the profiling points to change the profiling scope for filtering out useless profiling results to reduce the amount of profiling results. In the experiments, the required memory of profiling results using RMP for a browser application is smaller than debug class of Android 25 times, and the estimation error rate of execution time is proven lower than debug class 24 times because the debug class uses the entry time of java methods to calculate imprecise time results. Besides, the CPU and memory overhead of RMP are only 5% and 6.53% for the browsing scenario, respectively. From this evaluation, we found that the bottleneck of a browser is the web page drawing because the 2D graphical library does not use any hardware acceleration.

Keywords: time profiling, energy profiling, multi-resolution profiling, Android

Contents

List of Figures

List of Tables

Chapter 1. Introduction

For designing efficient embedded applications, two key design issues should be considered. First, the execution time of an embedded application should be optimized because they have to run on embedded devices with limited computing capability. Second, the energy consumption should be minimized because battery power is a bottleneck in embedded devices. Therefore, developers need to identify the hotspots in the program so that they can optimize their designs according to the analyzed results.

There were many timing and energy 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 resolution. For example, the PowerScope can only provide a coarse-grained profiling resolution, i.e., process level, to analyze the energy consumption of a process. Thus, users using PowerScope have the difficulty to identify the precise bottlenecks in the process. On the other hand, Gprof provides a fine-grained profiling resolution, i.e., function level, so users can analyze the time information of a process in more detail. However, users need to spend a lot of time to analyze logs of the detailed time information to identify performance bottlenecks.

Therefore, some tools adapted the multi-resolution profiling method [3-7] to provide more flexible analyses. These tools can analyze the fine-grained profiling traces and show results from a coarse-grained resolution to a fine-grained resolution of detail which can help users to easily analyze the bottlenecks. However, these tools are not suitable to be applied to embedded systems since they cannot efficiently store all logs of fine-grained profiling data due to the limited memory space in the embedded systems. Therefore, a novel multi-resolution profiling solution is required to help developers to profile their applications on the embedded system with limited log space.

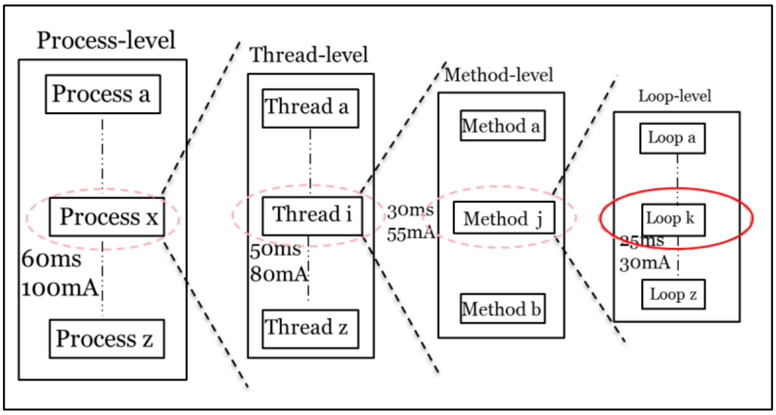


Figure 1. The profiling concept of RMP.

In this thesis, we design a new multi-resolution profiling solution and implement it, named reconfigurable multi-resolution profiling (RMP). It can efficiently profile execution time and energy consumption by using limited log space. The profiling concept of RMP is shown as Figure 1. The RMP profiles the execution time and energy consumption by changing profiling resolutions, such as process level, thread level, function level (method level) and loop level. For example, when the users start to profile an application, they can start from a specific coarse-grained resolution (e.g. process level). After users analyze the profiling results at this resolution, they may find some bottlenecks, such as the process x in Figure 1. The users can zoom in to profile the process at a more fine-grained resolution (e.g. thread level). When users identify that the thread i is the bottleneck of this process, they can zoom in a more detailed resolution again. Finally, the users can identify the bottlenecks of their application at loop k through the same zoom-in profiling process. The users can identify the actual bottlenecks in the application with small log space because the profiling process only profiles the necessary parts at each resolution.

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 the LTTng and the RMP is that the probes in RMP can be configured to change the profiling scope. To profile the energy consumption, we used an enhanced approach of Battery Use [8] to collect the energy information. It rebuilds the estimation model and power table to estimate the process level energy consumption. However, the resolution of time and energy consumption results may be not equal. Therefore, we will provide an approach to correlate timing and energy consumption results when the profiling resolution is finer than the process level.

The rest of this thesis is organized as follows. Chapter 2 discusses related work of time and energy profiling and the architecture of Android application. We describe the architecture of RMP and give an example run with a flow chart in Chapter 3. Chapter 4 shows the system implementation of RMP on an Android device. Chapter 5 presents the experimental environment and discusses evaluation results. Finally, Chapter 6 concludes the thesis and offers directions of future work.

Chapter 2. Background

The chapter first describes and compares various tools for time and energy profiling, and then briefs the architecture of Android applications.

2.1 Profiling Tools

Finding correct performance and energy bottlenecks is important to raise application's effectiveness and more easily debug. Profiling tools are usually used to identify the hotspots of an application to find the bottlenecks. A hotpot is a region of a program where a significant amount of computation or energy consumption occurs. Therefore, the first step to solve bottlenecks is use profiling tools to identify hotspots. Commonly used time and energy profiling tools are introduced as follows.

Time Profiling

Existing time profiling techniques can be divided into two categories: instrument-based and sampling-based. 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 Tracer (KFT) [9] use the compiler-assisted capabilities, such as –pg flag and –finstrument-functions flag in GNU compiler collection (GCC), to automatically instrument profiling points at entry and exit of every function. However, they can only provide the function level profiling because they only 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. The log event will be recorded when the probe is turned on and will be ignored when the probe is turned off. Therefore, the profiling resolutions depend on the location of instrumentation points.

Debug class [4], an Android built-in java class, provides a way to create log and trace the execution of an Android application. The source code of applications must be instrumented with specific code. TraceView [10] can analyze the log and show the execution information from process level to method level.

Sampling-based profiling tools utilize performance counters, presented in most modern CPUs, to record program execution information, such as program counter (PC), and CPU-related events, such as cache misses. Then the results can be correlated with the structure of source code. Representative tools include Oprofile [5], HPCToolkit [6] and Intel VTune [7]. 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.

Energy Profiling

Existing energy profiling can be divided into three categories: simulation approach, measurement approach, and estimation approach. Simulation approach creates virtual hardware platforms to simulate energy consumption behavior for energy profiling [11-12]. However, the energy optimization is not suitable by using simulators, because their accuracy is not high and the profiling has to work on virtual platforms, not the real platforms.

Measurement approach measures energy consumption with digital power meters directly. 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 information. The platform also collects some system information, such as program counter (PC) and process identifier (PID), for the sampling period. The energy consumption can be attributed to each function of a process according to recorded the PCs and PIDs. PowerScope [1] is the most famous tool for discovering the energy bottlenecks at the function level. ePro [13] integrates energy and performance sampling-based profiling into a convenient tool with user interface (UI) at the function level.

Estimation approach counts the requests of hardware components for each process. The amount of requests can be translated into energy consumption using the energy estimation model which includes the estimation formula and the power table. pTop [14] estimates component-wide energy consumption for each process and provides programming interface for designing energy-aware applications.

Battery Use has been embedded in Android system from version 1.6. During system booting, the “battery info” service takes responsibility for counting the requests of hardware components. Battery Use utilizes the inter process communication (IPC) to pull the data from the battery info service. An enhancement of Battery Use [8] 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%. This enhancement of Battery Use also provides the information of energy consumption for each process.

PowerSpy [15] is a hybrid estimation approach. It not only counts the requests of hardware components but also collects the thread IDs. In the analysis phase, the data will be translated into energy consumption and attributed to threads of a process according to the thread IDs.

Summary

Table 1 and Table 2 summarize the above discussed time and energy profiling tools, respectively. Gprof, 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 the fine-grained profiling trace at run time and show multi-resolution profiling 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 thesis, we will propose an approach to solve this limitation.

Table 1. Comparison of time profiling tools.

	Name	Profiling method	Profiling resolution	Features
Single-resolution profiling	Gprof[2]	Instrumentation	Function	● Instrument by compiler ● Interleaved logs problem
	KFT[9]	Instrumentation	Function	● Instrument by compiler ● Provide complete call graph
Multi-resolution profiling	LTTng[3]	Instrumentation	Process-statement	● Low overhead ● Not support java ● Friendly GUI
	Debug class[4]	Instrumentation	Process – method	● Android built-in ● Friendly GUI – Traceview [10]
	Oprofile[5]	Sampling	Process – statement	● Android support ● Not support java
	HPCToolkit[6]	Sampling	Process – loop	● Hardware dependency ● Friendly GUI
	Intel VTune[7]	Sampling	Thread- statement	● Hardware dependency ● Friendly GUI

Table 2. Comparison of energy profiling tools.

	Name	Profiling method	Profiling resolution	Features
Single-resolution profiling	PowerScope[1]	Measurement	Function	● Map energy consumption to program structure ● Sampling period : 1.6ms – low accuracy
	ePro[13]	Measurement and Sampling for time	Function	● Performance and energy profiling ● Friendly GUI
	pTop[14]	Estimation	Process	● Base on power table and estimation formula
	BatteryUse[8]	Estimation	Process	● Similar pTop ● Built-in Android
	PowerSpy[15]	Estimation	Thread	● Similar pTop ● On Windows os

2.2 Architecture of Android Applications

This section briefs Android applications, application components which are essential building blocks of an Android application and components lifecycles.

Android applications

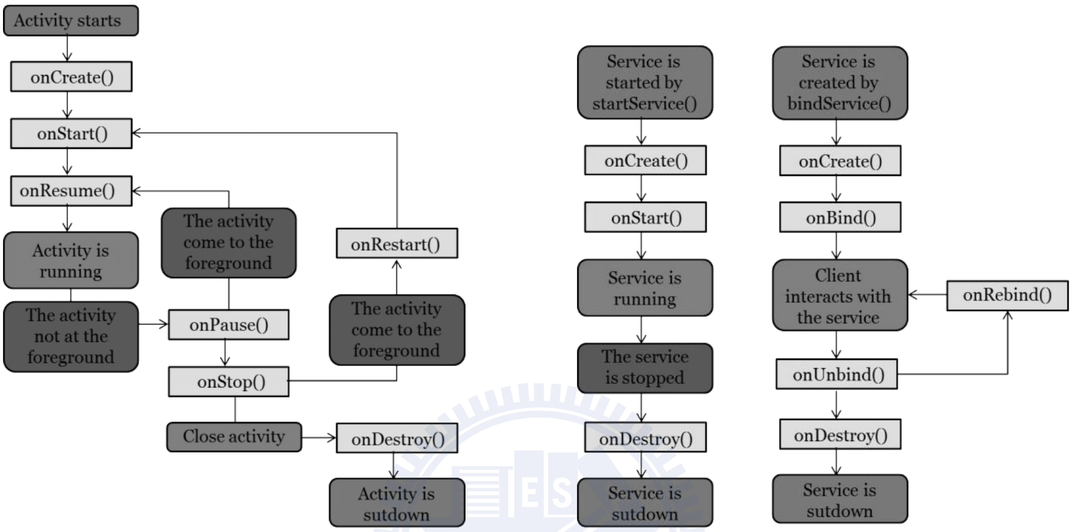
All user-visible applications on Android are written in java programming language. The source code, resource file, and other data of an application are compiled into an archive file with an .apk suffix. A single .apk file is used to install the application for Android devices. When the application is executed on Android, the Zygote which is the applications manager makes a new Linux process for the application. The process will be shutdown when the application is no longer needed or when the system must recover memory for other applications. In addition, each process has its own isolated Dalvik virtual machine (VM) which is derived from Java VM.

Application Components

There are four types of application components. First, an *activity* is the most common component in an Application. It provides a UI to accept user’s operation. For example, a browser application might have an activity that shows a web page, another activity to add a new bookmark, and another activity for searching the bookmarks. Second, a *service* component is a background program which performs long-running operations or to perform work for remote processes. For example, there are some users want to play music in a long time and operate other applications at the same time. In this case, the music player must be a service component which can run at the background and avoid bothering users. Third, a *content provider* is a manager of the application data. You can store the data to the SQLite database or other storage locations and set the data to be shared or private. If the data are shared, other applications can query or modify the data by the content provider. Fourth, a *broadcast receiver* is a component that responses the system’s broadcast messages. For example, the system broadcast a message to announce the power of system at the low battery state. The system’s applications can receive this message and response some actions.

Components lifecycles

As mentioned above, each types of application component have different functions and have different lifecycles too. The lifecycle of a content provider includes read/write and query the data. The lifecycle of a broadcast receiver is just to receive messages when the system broadcast messages. The lifecycles of activity and service are shown as Figure 2.



(a) Lifecycles of activity. (b) Lifecycles of service.

Figure 2. Lifecycles of application components.

The whole lifetime of activity is from onCreate() state to onDestroy() state. For example, if you have an activity to watch a video, it might call onCreate() to create a thread to play and then call onDestroy() to stop the thread. From onStart() state to onStop() state is called visible phase. During this phase, the activity is ready shown on screen to interact with users. And from onResume() state to onPause() state is called foreground phase. During this phase, the activity is in front of all activities on screen and interact with users. This phase of each activity may run serveral times, because there are only one activity can display its UI on screen and interact with users in the same time on Android devices.

The lifecycle of a service is similar to activity which is from onCreate() state to onDestroy() state, but the service doesn't have the visible and foreground phases because it doesn’t provide the user interface. A service can start from two forms with different lifecycles. First, when an application component start it by calling startService() to perform a single operation and doesn’t return a result to the caller. Second, when an application binds to it by calling bindService() to send requests and receive results.

Chapter 3. Reconfigurable Multi-Resolution Profiling Approach

This chapter describes the methodology of RMP. Section 3.1 briefs its concept and Section 3.2 describes its architecture. Finally, we present the profiling flow of RMP in Section 3.3.

3.1 Concept

In general, if users want to profile the execution time of an application by using limited log space in the embedded system, they can instrument some profiling points into application source code and framework source code (ex. Android framework) several rounds to collect time information at different profiling scope for achieving the same effect as RMP. However, they must spend a lot of time to recompile the application and the framework when they reinstrument other profiling points for changing the profiling scope. Therefore, RMP instruments all necessary profiling points into the application and the framework source code in the beginning. All profiling points are controlled according to the user configurations without recompiling the application and the framework. The energy profiling approach is based on an enhanced approach of Battery Use, which can estimate the energy consumption of processes by estimation formulas and power tables with the error rate below 10%.

3.2 Architecture

Figure 3 depicts the architecture of RMP which consists of five components, including instrumentation component, control and display component, time profiling component, energy profiling component, and logs correlation component. For changing profiling scope without recompiling, we design the instrumentation component which analyze the source code to instrument necessary profiling points with the resolution definitions and give the profiling point information to users. The control and display component display the profiling point information and the profiling results for users. Then users can control profiling scope according the resolution definitions and the previous profiling results. The time profiling component can only record the time information of profiling scope based on the configuration settings of the control and display component. The energy profiling component collects the battery information from the target system and gets the process level energy consumption results according to the battery information. Finally, the logs correlation component correlates time and energy consumption results, because the energy consumption results are profiled from process level and it cannot be analyzed from fine-grained resolutions. Thus the logs correlation component correlates the process level energy consumption with fine-grained time profiling results for analyzing energy bottlenecks in fine-grained resolutions.

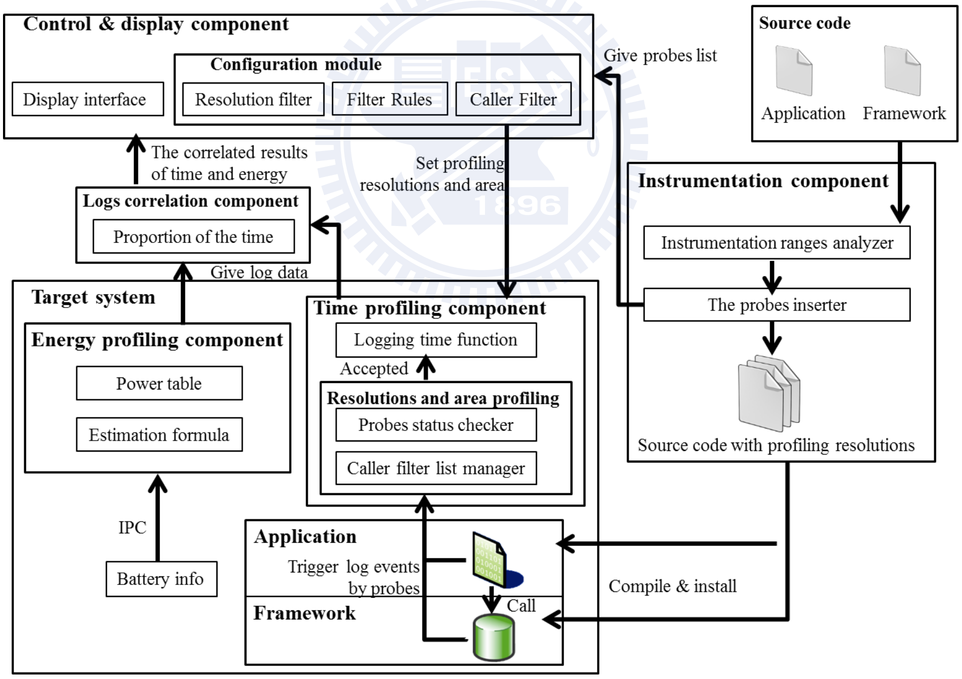
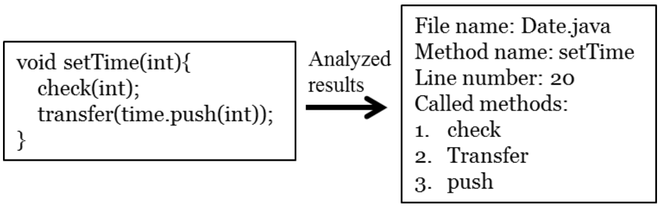


Figure 3. Architecture of RMP.


```
void setTime(int){
   check(int);
   transfer(time.push(int));
}
```

File name: Date.java
Method name: setTime
Line number: 20
Called methods:
1. check
2. Transfer
3. push

Figure 4. Example of source code analysis.

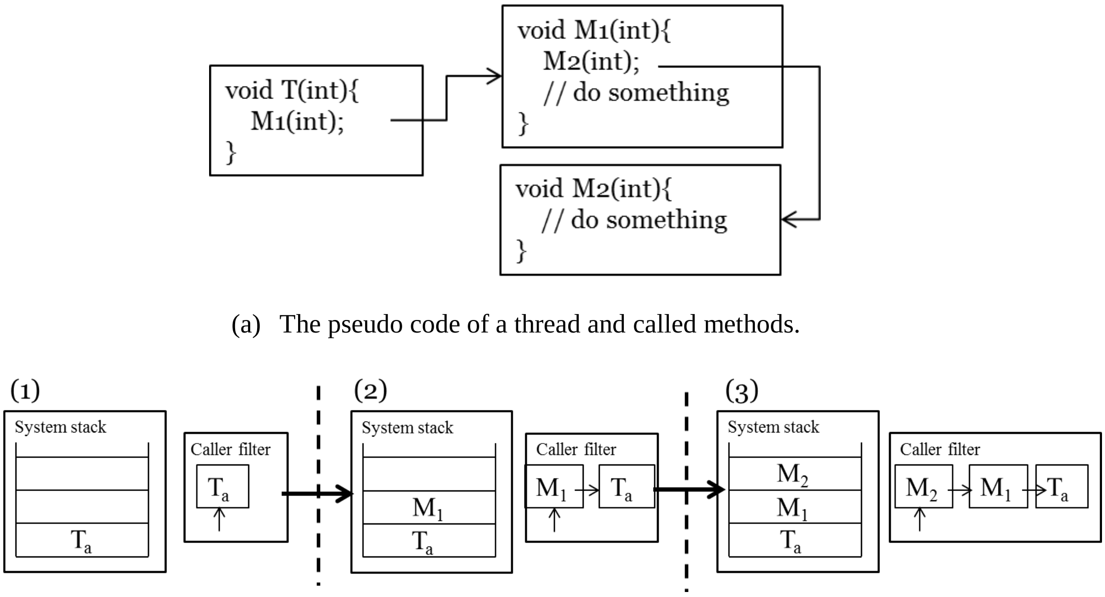
Instrumentation component

In RMP, the proposed instrumentation method instruments profiling points to the whole necessary locations of the application and the framework in the beginning. However, only some framework methods will be called by an application. If we instrument all profiling points to the whole framework, some profiling points will be useless, and they may cause extra unnecessary overhead for other applications. Therefore, the instrumentation range of the application should be identified before we instrument profiling points in the framework. For example, in Figure 4, the instrumentation ranges analyzer scans the application source code and the framework source code to identify the method's location and the used application and framework methods. The methods of the application and the methods of the framework which may be called in the application are defined as the instrumentation range.

When the instrumentation range is identified, we can start to instrument profiling points into the application and framework source code. We instrument profiling points at the entry and exit of every method and every method's loop because the execution time of each method and loop can be estimated by the entry time and the exit time. The profiling points of a method or a loop map to a probe which is a control unit when users control profiling scope in the control and display component. In instrumentation component, each probe is recorded in a probe list with its probe name, resolution definition and status. The resolution definition can be recognized by specific methods functionality (ex. the run() method can be seem as thread level). Then, the application source code and framework source code can be compiled and installed on the target system to do multi-resolution profiling without recompiling when users change the profiling scope.

Control and display component

The control and display component read and show the probes information of probes list or the correlated profiling results to users on display interface. Then users can turn on or turn off the probes to control profiling scope by the configuration module. The profiling scope is according from the settings of profiling resolutions and profiling area. The profiling resolutions are the language-based levels, such as process, thread, function (method) and loop. Users can zoom a coarse-grained resolution into a fine-grained resolution for analyzing bottlenecks and saving the log space by the resolution filter, which turn the probes on or off according the users settings with the resolution definition of probes. The profiling area is defined as the set of probes which will be profiled. For example, when the profiling is working at the method level of a thread i, the profiling area is all method level probes in thread i. Also when the profiling zooms in the method k of thread i, the profiling area includes all loop level probes in method k. However, a method may be executed by other threads which are not required to do the profiling. Therefore, we design a caller filter for users to solve the problem. Users can set some root points in the caller filter, and then only some methods that are called from the root points can be profiled. To identify what methods are called from the root points, the caller of methods will be dynamically records into a caller filter list at runtime.



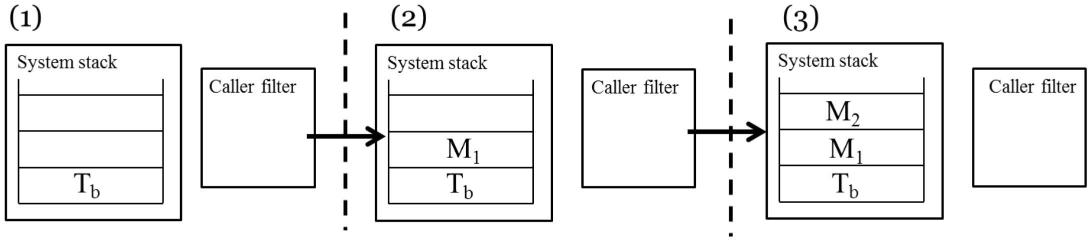
```
void T(int){
   M1(int);
}

void M1(int){
   M2(int);
   // do something
}

void M2(int){
   // do something
}
```

(a) The pseudo code of a thread and called methods.

(b) T_a is the root point.



(c) T_b is not the root point.

Figure 5. Example of a caller filter.

The example of the caller filter is shown as Figure 5. We suppose methods 'M_1' and 'M_2' are called by thread 'T_a' and 'T_b' in Figure 5(a). We set 'T_a' as our root point in Figure 5(b1), and when 'T_a' is executed, it will be recorded into the caller filter list, which is maintained in the time profiling component. In Figure 5(b2), 'M_1' will be recorded into the caller filter list when it is called by 'T_a' because the caller of 'M_1' is 'T_a' which has been recorded in the list. Also 'M_2' will be also recorded into the caller filter list too in Figure 5(b3), because its caller is 'M_1' who has been recorded in the list. After that, 'M_2' will be removed from the caller filter list when it is finished. Then 'M_1' will resume until it is finished and removed from the caller filter list. Finally, 'T_a' will be removed from the caller filter list when it is finished. Therefore, when each of them is recorded in the caller filter list, it will be profiled. There is a counterexample in Figure 5(c1), 'T_b' is not recorded into the caller filter list because it is not our root point. Thus 'M_1' and 'M_2' are not recorded into the caller filter list in Figure 5(c2) and Figure 5(c3), respectively, because their callers are not recorded in the list. Therefore, these methods called by 'T_b' are not profiled.

In addition, the caller filter can not only solve the above problem, but also can be used to control profiling area when zoom in other resolutions. For example, when users zoom in the loop level of 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 some filter rules users specified, such as execution time over than one sec, energy consumption over than 20 μAh, and executing times over than five, can also help users to limit profiling area according to profiling results. These specific filtering rules can help users exclude some log events which users do not want to profile for reducing the log size.

Time and energy profiling components

The time profiling component checks the probes status (on or off) and manages the caller filter list based on the configuration settings of configuration module before it records the time data. When the application executing 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 is on and the caller of probes have been recorded in the caller filter list, the log events will be accepted and the system time will be recorded into log space in memory, which amount can be set by users. The logs will be removed from memory to other large-storage components when the log space of time profiling component is out of bound or the profiling has ended. The energy profiling component based on the enhanced approach of Battery Use which uses IPC to collect the battery information and calibrate it to the process level energy consumption results by power tables and estimation formulas.

Logs correlation component

The time and energy profiling component generates time and energy consumption results to users. However, the results of energy consumption are profiling from process level, and it cannot be analyzed from fine-grained resolutions. Therefore, we need to get a fine-grained energy consumption result. Thus we try to correlate the process level energy consumption with fine-grained time profiling results before we correlate them, we should know that the energy consumption can be divided into two parts. The first part is the energy consumption of asynchronous components, such as Wi-Fi and Bluetooth. These components receive requests from an application to do some tasks asynchronously and return results to the application when the tasks have been finished. Therefore, their energy consumption results cannot be correlated with time results of the application because the execution of these components and execution of the application are independent. Another part is the energy consumption of synchronous components, such as CPU and memory. These components are the major contributors of the execution time of applications, so their energy consumption can be correlated with time results of applications.

The Battery Use provides these two types of energy consumption. The consumption of asynchronous components is calibrated to each components and the consumption of synchronous components is calibrated to each process. Therefore, we can directly use the energy consumption of a process (an application) to correlate with our time results of an application. The correlation between time and energy can be written as

$$T_{i,j} = \frac{T_{p_{i,j}}}{T_{p_{i,1}} + \cdots + T_{p_{i,n_i}}}$$

$$E^{syn}_{p_{i,j}} = \begin{cases} E^{\;syn}_{\;process} \times T_{i,j},\; i = 2 \\ E^{syn}_{p_{i-1,k}} \times T_{i,j},\; i > 2 \end{cases}$$

where *i* is the profiling resolution, *j* is the probe, *n* is the number of probes in resolution i, *k* represents the selected probe in upper resolution. The $T_{p_{i,j}}$ is the execution time of probe j at resolution *i* and the $E^{syn}_{p_{i,j}}$ is the energy consumption of probe j at resolution *i*.

If process level is the first resolution (i = 1), the energy consumption of a probe in next resolution (i=2) is proportion of the execution time of the probe in the current resolution, i.e., $E^{\;syn}_{\;process}$. When the profiling resolution is a fine-grained resolution (i

> 2), the energy consumption of a probe can be calculated from the energy consumption of the upper-resolution probe by proportion of the execution time. For example as shown in Figure 6, we assume the energy consumption of a process is 10 mA. The energy consumption of the two threads in the process can be directly calculated by proportion of the execution time of the process. And the energy consumption of the two methods in thread k can be calculated from 7 mA by proportion of the execution time of each method, too.

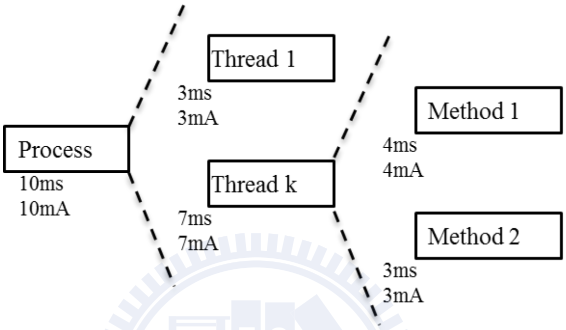


Figure 6. Example of logs correlation.

3.3 Flow Chart

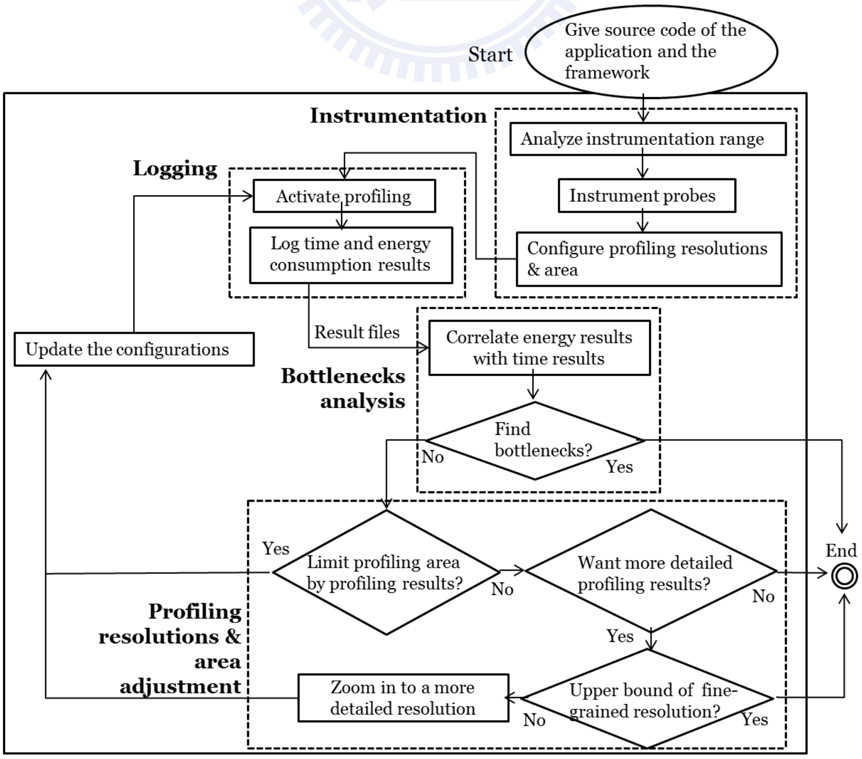

Start
Give source code of the application and the framework
Instrumentation
Analyze instrumentation range
Instrument probes
Configure profiling resolutions & area
Logging
Activate profiling
Log time and energy consumption results
Result files
Update the configurations
Correlate energy results with time results
Bottlenecks analysis
Find bottlenecks?
No
Yes
Yes
Limit profiling area by profiling results?
No
Want more detailed profiling results?
No
End
Yes
Profiling resolutions & area adjustment
Zoom in to a more detailed resolution
Upper bound of fine-grained resolution?
No
Yes

Figure 7. Flow chart of RMP.

Figure 7 depicts the main flow chart which contains four profiling phase, including instrumentation, logging, bottlenecks analysis, and profiling resolutions & area adjustment. In instrumentation phase, when users give source code of the application and the framework for profiling, we 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. Next, users can configure the profiling resolution and area to start profiling by the control and display component. In logging phase, the time and energy profiling results will be stored in memory and written to result files at the end. The logs correlation component can read result files and correlate them by the proportion of execution time to analyze bottlenecks. In profiling resolutions & area adjustment phase, users can adjust some filter rules and the caller filter to limit profiling area with the previous run-time profiling results. They can zoom in a more detailed resolution by the resolution filter and caller filters, if necessary. After doing the logging phase, bottlenecks analysis phase and profiling resolutions & area adjustment phase several rounds, users may find the bottlenecks of an application on a resource-constrained embedded system without recompiling the application and the framework.

Chapter 4. Implementation on Android

This chapter details the implementation of the reconfigurable multi-resolution profiling approach on the Android Dev Phone 1 (Dev 1) which is taken as the device under test (DUT). The implementations of the instrumentation and time profiling components are introduced in Section 4.1, 4.2. The control and display component and logs correlation component are introduced in Section 4.3.

4.1 Instrumentation Component

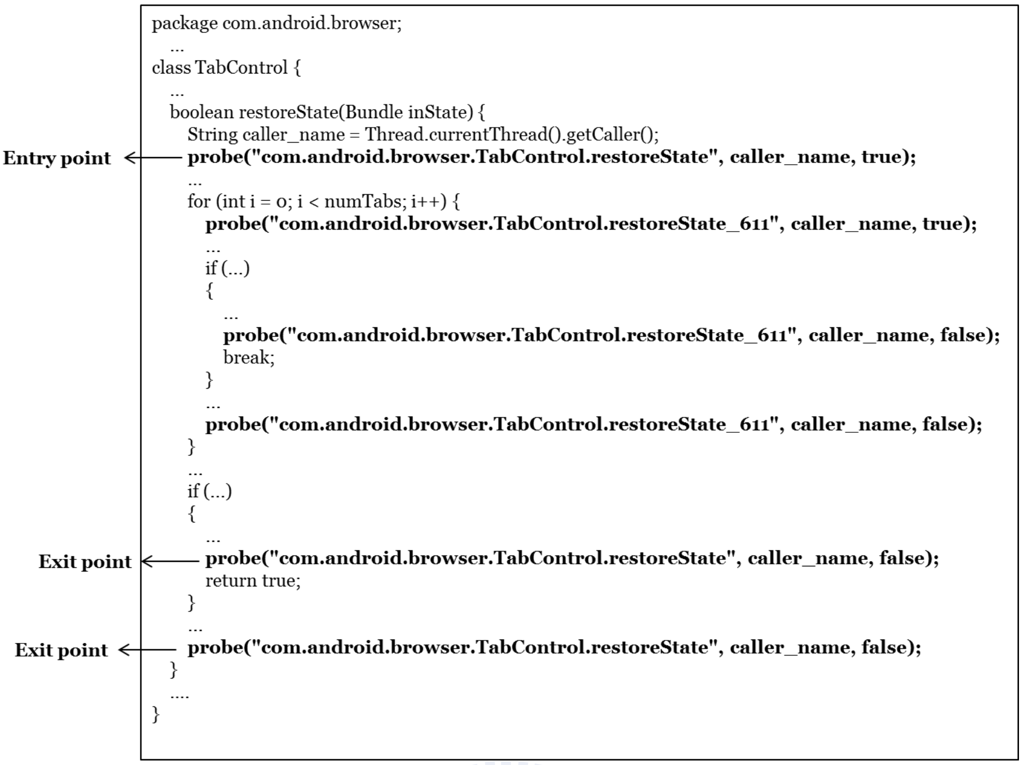
The instrumentation range analyzer can analyze the methods location and what framework methods are called by an application. Then, the probes inserter can instrument profiling points into the application source code and the framework source code based on the results of instrumentation range analyzer.

The instrumentation flow includes three phases. First, we use Jindent [16], which is a source code formatter tool, to unify the source code format of the application and the framework, because the unification format of source code can minimize the complexity of our instrumentation algorithm. For example, if the declaration of a method and its left curly brace are written at the same line, our entry profiling point can be instrumented at the next line directly. If the left curly brace is written into the next line of method declaration, the entry profiling point need to be instrumented at right side of left curly brace or next line of left curly brace. In this implementation, we can just consider the same line case when the source code has been formatted.

Second, we use ctags [17] to get the location of the application methods and the framework methods. The results of ctags include methods name, file name of methods and line number of methods in the file. And 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 system, such as Windows and Linux.

Finally, we implement another automatic script to instrument entry and exit profiling points into the application methods and the framework methods with the resolution definition. We define four profiling resolutions for Android application in this implementation. First, the process level is the total execution time of all 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 bottleneck 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 in onCreate() state to 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 visible phase. The execution time of a *service* can be estimated by the period from onCreate() state to onDestroy() state, and execution time of a *broadcast receiver* can be estimated by the time of onReceive(). However, the *content provider* doesn’t have the independent execution time, because it called by other components in general. Therefore, the content provider’s execution time has been included in other components.

Second, the thread level is the execution time of run( ) method which needs to be implemented in a thread class. This resolution is used to analyze the bottleneck of threads in a process (If the process has threads). Third, the method level is the execution time of each normal method. This resolution is used to analyze the bottleneck of methods in a thread. Finally, the loop level is the execution time of each loop. This resolution is used to analyze the bottleneck of loops in a method.



```
package com.android.browser;
   ...
class TabControl {
   ...
   boolean restoreState(Bundle inState) {
      String caller_name = Thread.currentThread().getCaller();
      probe("com.android.browser.TabControl.restoreState", caller_name, true);
      ...
      for (int i = 0; i < numTabs; i++) {
         probe("com.android.browser.TabControl.restoreState_611", caller_name, true);
         ...
         if (...)
         {
            ...
            probe("com.android.browser.TabControl.restoreState_611", caller_name, false);
            break;
         }
         ...
         probe("com.android.browser.TabControl.restoreState_611", caller_name, false);
      }
      ...
      if (...)
      {
         ...
         probe("com.android.browser.TabControl.restoreState", caller_name, false);
         return true;
      }
      ...
      probe("com.android.browser.TabControl.restoreState", caller_name, false);
   }
   ....
}
```

Figure 8. The method level and loop level instrumentation examples.

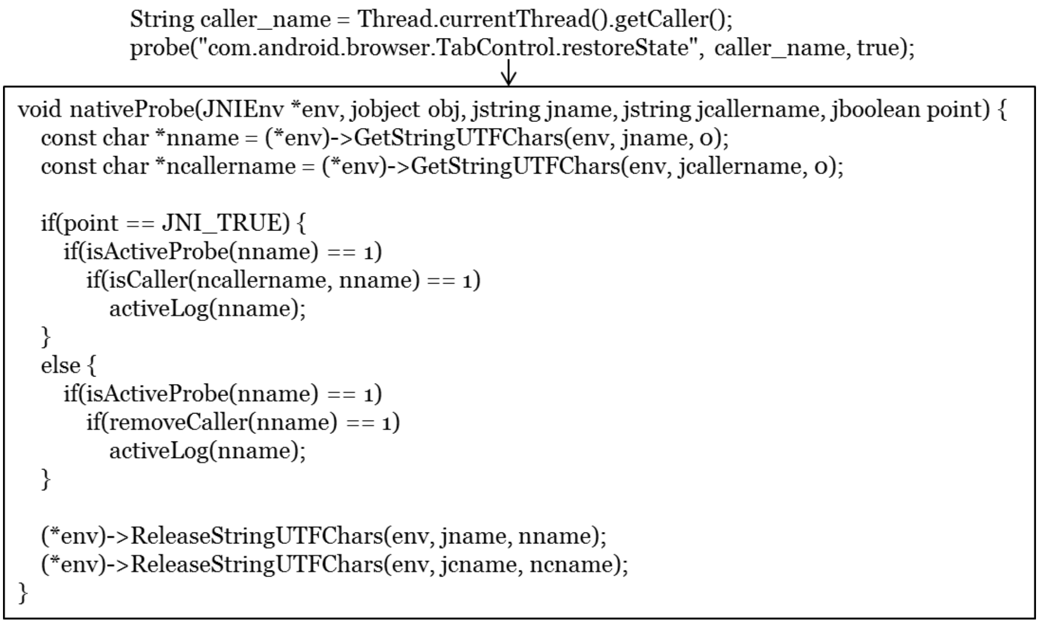
There is an instrumentation example shown as Figure 8. The entry points of thread level and method level are instrumented at the next line of method declaration and the exit points are instrumented before the finish points, such as right curly brace of method declaration, return syntax and throw syntax. The entry and exit profiling points of a method map to the same probe with the profiling resolution definition and the probe name for users to configure the profiling resolutions and area in control and display component. The probe name is composed of the package name, class name and method name.

The entry points of loop level are instrumented to the next line of loop declaration, such as for and while, and the exit points are instrumented before the right curly brace of loop syntax and break syntax. And the loop level profiling points map to its loop level probes in a method. However, the probe name is same with other resolution because the method name, package name and class name are the same in a method. Therefore, the line number of the loop will be concatenated to the original method level probe name.

4.2 Time Profiling Component

When the profiling points are triggered, the time profiling component receives log events, checks probes status and manages caller filter list based on the configuration settings of configuration module before it record the time data into the memory. These actions need to be implemented efficiently because they may influence the accuracy of time data if they spend too much time.

Therefore, the proposed implementation of the time profiling component is written in C language, which has higher performance than java language. Unfortunately, the Android application and framework are written in java language, profiling points (written in java language) cannot call the time profiling component directly. So we use java native interface (JNI), a most effective method to call C program from Java side on Android [18], for profiling points to call the time profiling component. The time profiling component is implemented in a JNI native library and it will be loaded when the application and the framework start to execute. However, the application and the framework cannot load the same time profiling component when it implemented in a JNI native library because the application and the framework classes are executed by different class loader, and the same JNI native library cannot be loaded into more than one class loader. According to the official statement [19], the benefit of this limit is the name space separation which preserved in native library based on class loaders. The native library cannot easily mix classes from different class loaders. Therefore, we separate the time profiling component implementation into two parts (application part and framework part) to solve above problem.



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

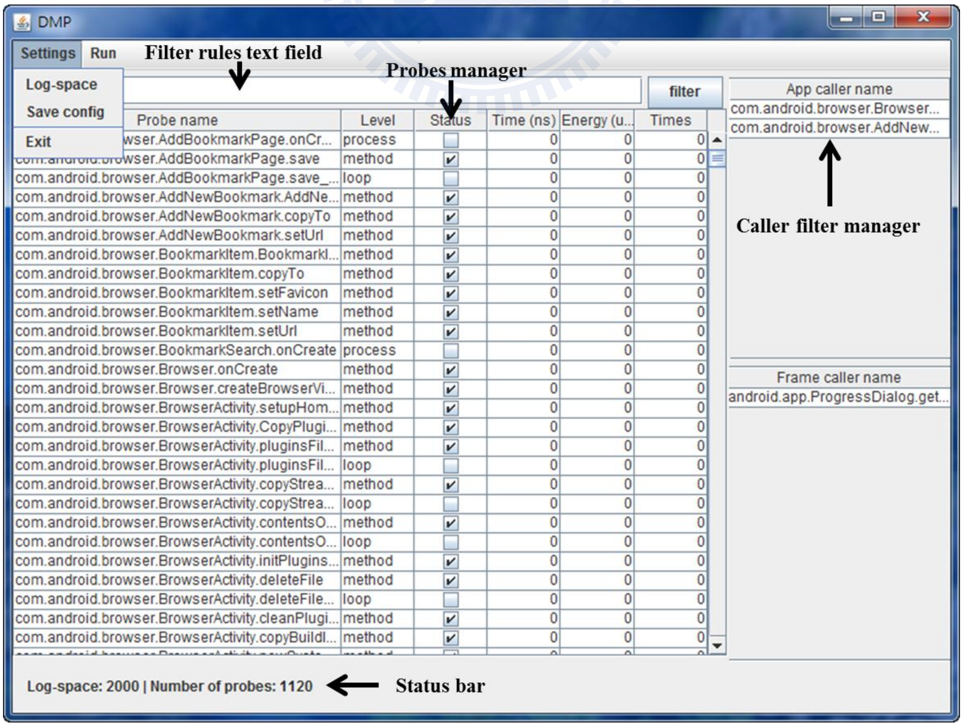
Figure 9. The example of calling the time profiling component.

Figure 9 shows an example of calling the time profiling component using JNI. The profiling points call the time profiling component through the JNI application programming interface and send its probe name, caller name and point attribute ('true' is entry point and 'false' is exit point) for checking. We can use the API, Thread.currentThread.getStackTrace(), to collect the stack information and select out the caller name from stack information. However, it spends too much execution time and we do not need to get whole stack information. We only need the caller name in stack. Therefore, we implement a new API, Thead.currentThread.getCaller(), to only return the caller name.

The time profiling component, either application part or framework part compares the probe name with the probes configuration settings which record the probes status from configuration module. When the time profiling component found the status of a probe is off, the log event is ignored. And when the status of a probe is on, the checking is passed and starts to check caller name. The caller name is used to compare with the head of caller list. But the time profiling component of framework part check the caller name is come from the target application before check the caller list, it needs to avoid accepting the log event which called by other applications. If the profiling point is entry point and the caller name is different with the head of caller list, this log event will be ignored. On the other hand, if the caller name is same with the head of caller list, then the name of profiling point will be added into the head of caller list and the log event will be accepted to record the entry time into log space in memory. If the profiling point is exit point and the caller name is different with the head of caller list, the time profiling component will ignore this log event. On the other hand, if the caller name is same with the head of caller list, then the head of list will be removed and the log event will be accepted to record the exit time.

When the log event accepted to record the time, time profiling component call the activeLog() API to get the nanosecond time by calling clock_gettime() API and store it into log space in memory. And the time data will be written into ram disk when the log space is out of bound or the profiling has ended.

4.3 Control & Display and Logs Correlation Components



Filter rules text field

Probes manager

Caller filter manager

Status bar

Probe name	Level	Status	Time (ns)	Energy (u...	Times
wser.AddBookmarkPage.onCr...	process	□	0	0	0
com.android.browser.AddBookmarkPage.save	method	☑	0	0	0
com.android.browser.AddBookmarkPage.save_...	loop	□	0	0	0
com.android.browser.AddNewBookmark.AddNe...	method	☑	0	0	0
com.android.browser.AddNewBookmark.copyTo	method	☑	0	0	0
com.android.browser.AddNewBookmark.setUrl	method	☑	0	0	0
com.android.browser.BookmarkItem.BookmarkI...	method	☑	0	0	0
com.android.browser.BookmarkItem.copyTo	method	☑	0	0	0
com.android.browser.BookmarkItem.setFavicon	method	☑	0	0	0
com.android.browser.BookmarkItem.setName	method	☑	0	0	0
com.android.browser.BookmarkItem.setUrl	method	☑	0	0	0
com.android.browser.BookmarkSearch.onCreate	process	□	0	0	0
com.android.browser.Browser.onCreate	method	☑	0	0	0
com.android.browser.Browser.createBrowserVi...	method	☑	0	0	0
com.android.browser.BrowserActivity.setupHom...	method	☑	0	0	0
com.android.browser.BrowserActivity.CopyPlugi...	method	☑	0	0	0
com.android.browser.BrowserActivity.pluginsFil...	method	☑	0	0	0
com.android.browser.BrowserActivity.pluginsFil...	loop	□	0	0	0
com.android.browser.BrowserActivity.copyStrea...	method	☑	0	0	0
com.android.browser.BrowserActivity.copyStrea...	loop	□	0	0	0
com.android.browser.BrowserActivity.contentsO...	method	☑	0	0	0
com.android.browser.BrowserActivity.contentsO...	loop	□	0	0	0
com.android.browser.BrowserActivity.initPlugins...	method	☑	0	0	0
com.android.browser.BrowserActivity.deleteFile	method	☑	0	0	0
com.android.browser.BrowserActivity.deleteFile...	loop	□	0	0	0
com.android.browser.BrowserActivity.cleanPlugi...	method	☑	0	0	0
com.android.browser.BrowserActivity.copyBuildI...	method	☑	0	0	0

App caller name
com.android.browser.Browser...
com.android.browser.AddNew...

Frame caller name
android.app.ProgressDialog.get...

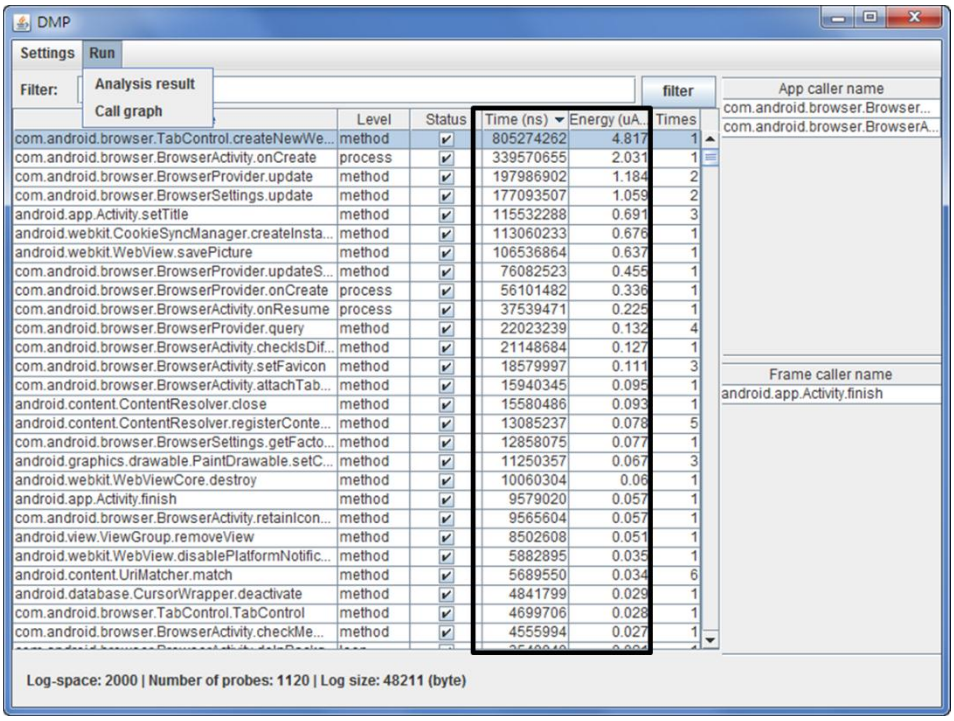
Log-space: 2000 \| Number of probes: 1120

Figure 10. The control and display component.

The Figure 10 shows an example of control and display component. The probes manager block read the probes list to show the probes name, resolution definition of each probe. It also gives an optional interface, checkbox component, to each probe for setting the probe status to control profiling resolutions and area. The profiling resolutions and area can be controlled quickly by writing some filtering rules into the filter rules text field, such as level = method, time >= 2000 and times > 2.

The probe will be added into caller filter manager block when users double click a probe in the probes manager. The caller filter manager block is separated into two parts because the time profiling component is implemented into the application part and the framework part. Therefore, the application probes are added into the application part and the framework probes are added into the framework part. The content of caller filter manager block will be the settings for managing caller filter list in the time profiling component. The final block is status bar. When users profile an application, it provides some information to users, such as log space and number of probes. In addition, the menu of the control and display component provides some functions for users to edit the log space size, save the configuration settings to devices and run the process of the logs correlation component.

The Figure 11 shows an example of correlated profiling results. The probes manager block read the time and energy consumption results when users run the analysis results function in the menu. The logs correlation component can automatically calculate the execution time of each probe by the entry time and exit time, and use proportion of the time method to correlate energy consumption results with time results for analyzing.



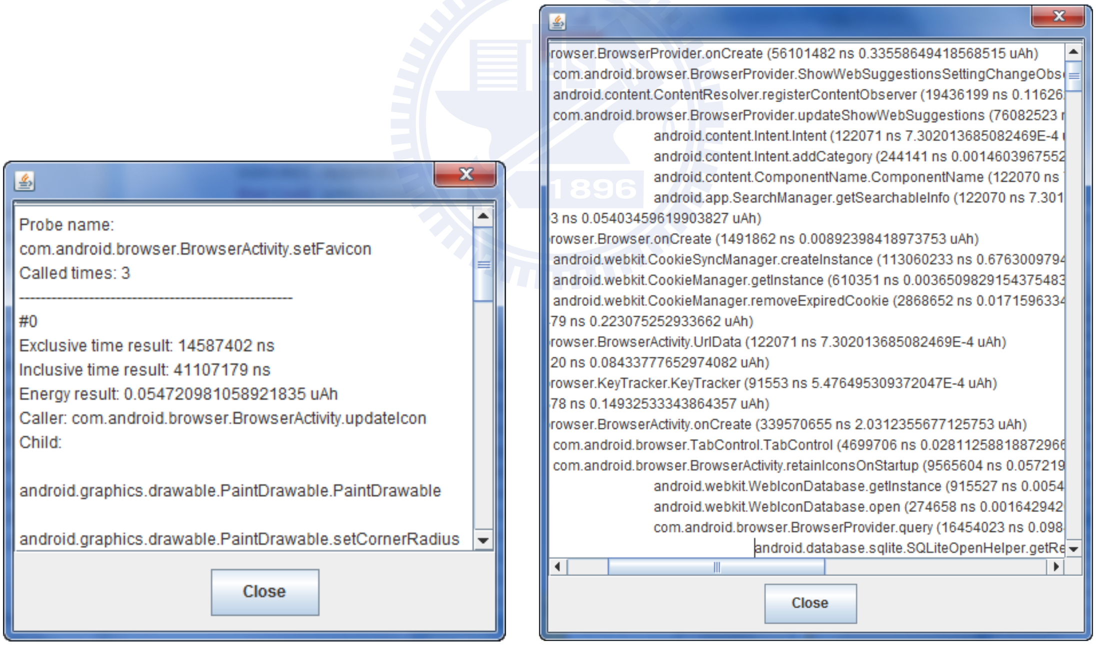
	Level	Status	Time (ns)	Energy (uA...	Times
com.android.browser.TabControl.createNewWe...	method	☑	805274262	4.817	1
com.android.browser.BrowserActivity.onCreate	process	☑	339570655	2.031	1
com.android.browser.BrowserProvider.update	method	☑	197986902	1.184	2
com.android.browser.BrowserSettings.update	method	☑	177093507	1.059	2
android.app.Activity.setTitle	method	☑	115532288	0.691	3
android.webkit.CookieSyncManager.createInsta...	method	☑	113060233	0.676	1
android.webkit.WebView.savePicture	method	☑	106536864	0.637	1
com.android.browser.BrowserProvider.updateS...	method	☑	76082523	0.455	1
com.android.browser.BrowserProvider.onCreate	process	☑	56101482	0.336	1
com.android.browser.BrowserActivity.onResume	process	☑	37539471	0.225	1
com.android.browser.BrowserProvider.query	method	☑	22023239	0.132	4
com.android.browser.BrowserActivity.checkIsDif...	method	☑	21148684	0.127	1
com.android.browser.BrowserActivity.setFavicon	method	☑	18579997	0.111	3
com.android.browser.BrowserActivity.attachTab...	method	☑	15940345	0.095	1
android.content.ContentResolver.close	method	☑	15580486	0.093	1
android.content.ContentResolver.registerConte...	method	☑	13085237	0.078	5
com.android.browser.BrowserSettings.getFacto...	method	☑	12858075	0.077	1
android.graphics.drawable.PaintDrawable.setC...	method	☑	11250357	0.067	3
android.webkit.WebViewCore.destroy	method	☑	10060304	0.06	1
android.app.Activity.finish	method	☑	9579020	0.057	1
com.android.browser.BrowserActivity.retainIcon...	method	☑	9565604	0.057	1
android.view.ViewGroup.removeView	method	☑	8502608	0.051	1
android.webkit.WebView.disablePlatformNotific...	method	☑	5882895	0.035	1
android.content.UriMatcher.match	method	☑	5689550	0.034	6
android.database.CursorWrapper.deactivate	method	☑	4841799	0.029	1
com.android.browser.TabControl.TabControl	method	☑	4699706	0.028	1
com.android.browser.BrowserActivity.checkMe...	method	☑	4555994	0.027	1

App caller name
com.android.browser.Browser...
com.android.browser.BrowserA...

Frame caller name
android.app.Activity.finish

Log-space: 2000 | Number of probes: 1120 | Log size: 48211 (byte)

Figure 11. The correlated profiling results.



```
Probe name:
com.android.browser.BrowserActivity.setFavicon
Called times: 3
--------------------------------------------------
#0
Exclusive time result: 14587402 ns
Inclusive time result: 41107179 ns
Energy result: 0.054720981058921835 uAh
Caller: com.android.browser.BrowserActivity.updateIcon
Child:

android.graphics.drawable.PaintDrawable.PaintDrawable

android.graphics.drawable.PaintDrawable.setCornerRadius
```

```
rowser.BrowserProvider.onCreate (56101482 ns 0.33558649418568515 uAh)
 com.android.browser.BrowserProvider.ShowWebSuggestionsSettingChangeObs
 android.content.ContentResolver.registerContentObserver (19436199 ns 0.11626
 com.android.browser.BrowserProvider.updateShowWebSuggestions (76082523 n
               android.content.Intent.Intent (122071 ns 7.302013685082469E-4 u
               android.content.Intent.addCategory (244141 ns 0.0014603967552
               android.content.ComponentName.ComponentName (122070 ns 7
               android.app.SearchManager.getSearchableInfo (122070 ns 7.301
3 ns 0.05403459619903827 uAh)
rowser.Browser.onCreate (1491862 ns 0.00892398418973753 uAh)
 android.webkit.CookieSyncManager.createInstance (113060233 ns 0.6763009794
 android.webkit.CookieManager.getInstance (610351 ns 0.0036509829154375483
 android.webkit.CookieManager.removeExpiredCookie (2868652 ns 0.0171596334
79 ns 0.223075252933662 uAh)
rowser.BrowserActivity.UrlData (122071 ns 7.302013685082469E-4 uAh)
20 ns 0.08433777652974082 uAh)
rowser.KeyTracker.KeyTracker (91553 ns 5.476495309372047E-4 uAh)
78 ns 0.14932533343864357 uAh)
rowser.BrowserActivity.onCreate (339570655 ns 2.0312355677125753 uAh)
 com.android.browser.TabControl.TabControl (4699706 ns 0.02811258818872966
 com.android.browser.BrowserActivity.retainIconsOnStartup (9565604 ns 0.057219
               android.webkit.WebIconDatabase.getInstance (915527 ns 0.0054
               android.webkit.WebIconDatabase.open (274658 ns 0.001642942
               com.android.browser.BrowserProvider.query (16454023 ns 0.098
                              android.database.sqlite.SQLiteOpenHelper.getRe
```

(a) Information of each times.

(b) Call graph example.

Figure 12. The detailed information of probes.

However, the time and energy consumption results of probes are the average exclusive time results and the average energy consumption in probes manager block. Users may want to know the inclusive time results or the results (time and energy consumption) of each round. Therefore, users can click right mouse button on each probe to show the detailed information window, as shown in Figure 12(a). Users can know the exclusive time, inclusive time, energy consumption, caller and child methods (called methods) information of each round in this window because we instrument entry and exit profiling points to collect the execution time of each method and analyze it by the time relation of each method. And based on the caller and child methods information, we can provide a call graph easily too, as shown in Figure 12(b).

Chapter 5. Evaluation Studies

In this chapter, a testbed and some experiments are designed to evaluate the overhead and the accuracy of the RMP profiling for real scenarios. Moreover, we present the profiling results obtained from an off-the-shelf Android device for a user-perceptible scenario, browsing.

5.1 Testbed

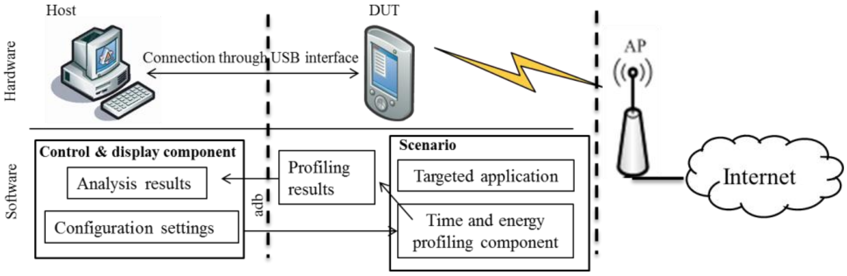


Figure 12. Testbed.

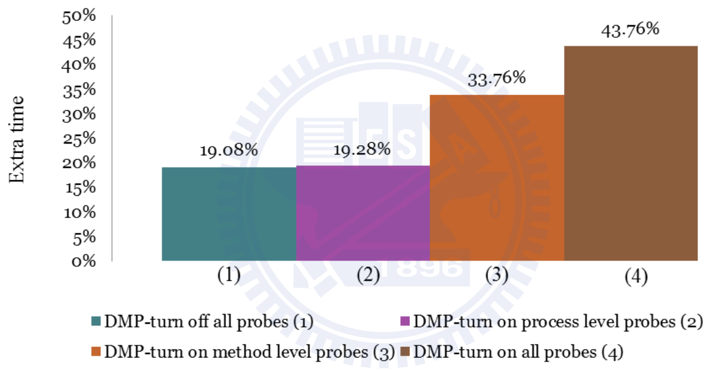
Figure 12 depicts the testbed. The experiments in this work were conducted on an Android-based DUT, Android Dev Phone1. The DUT is an Android-based smartphone which provides root permission for users to reinstall programs or whole system on the platform. The hardware of the phone is same with the HTC dream, and the phone can be regarded as a commercial product. The version of Android platform is Android open source project (ASOP) 1.6. Beside the DUT, a host machine provides users an operating interface to the DUT. The operating interface on the host is a client program, named Android debug bridge (adb), and the corresponding server on the DUT is adbd. The host connects with DUT through the USB interface to send the configurations or receive the profiling results by the operating interface. Finally, the IEEE 802.11g (Wi-Fi) network is accessible in the testbed environment.

5.2 Evaluation Scenario

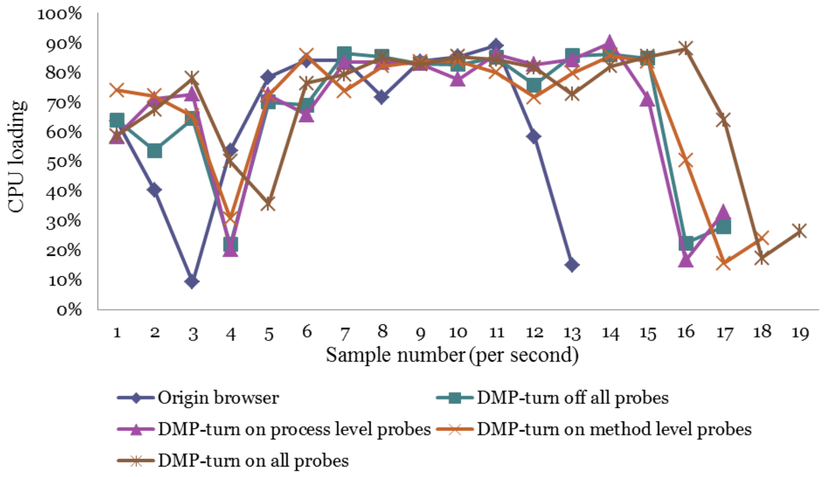
In this work, most evaluation experiments are based on the browsing scenario because the Android default browser spend a lot of time when it starts to load and shows the web page. Therefore, we profile the bottlenecks of the browser by RMP. The browsing scenario starts from the users touch the screen to start the browser and stops at the default page (Yahoo! mobile version, tw.m.yahoo.com) have been loaded and shown on the screen. One of the evaluation experiments is about the effects of log size. This experiment needs a lot of logs to do the experiment. However, the size of generated logs of the browsing scenario is not enough for the experiment. Therefore, we use another scenario, music playing, for this experiment because it will generate a lot of logs for updating the time information of music.

5.3 Experimental Results

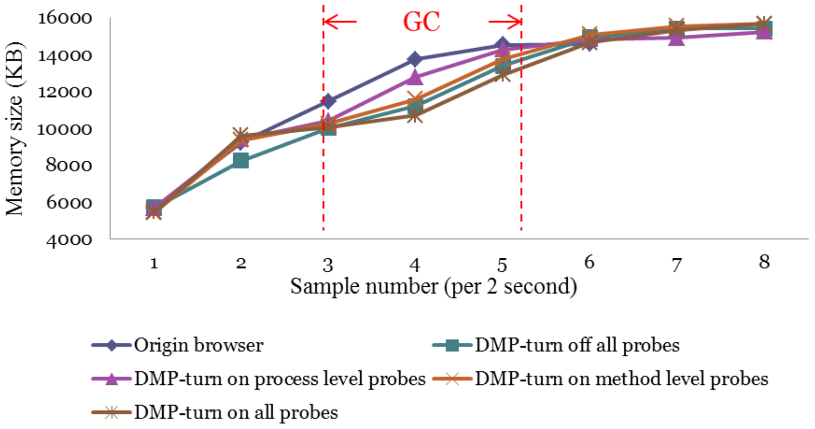
Experiment of overhead



(a) The overhead of execution time for RMP.



(b) CPU Loading.



(c) The overhead of memory.

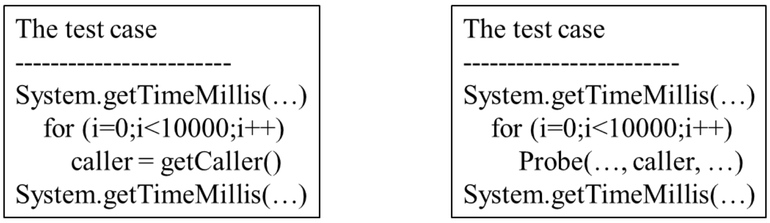
Figure 13. The overhead of RMP.

In this experiment, the overhead of execution time, CPU and memory are measured for the browsing scenario. Figure 13(a) shows that the execution time of the browser is extended 19.08% when we turn off all probes (not record any log events) and extended 43.76% when we turn on all probes (record all log events). The extended time is composed of three parts. First, each log event of probes needs to spend time to check probes status, caller filter list and store time results. Second, the VM needs to spend time to initial the time profiling component (JNI shared library) when it loads and initiates classes which are instrumented with probes. Third, the time profiling component stores time results in the memory and removes it from memory to other large-storage components when the log space is out of bound. Therefore, time profiling component will allocate memory and free memory several times. These actions will trigger the VM spends time to do garbage collection.

Fig 13(b) depicts the CPU loading of the browsing scenario. From the figure, the ups and downs of CPU loading are similar because the computation of the time profiling component does not need too much CPU resource. Therefore, the average of extra overhead on CPU is 5% when we turn on all probes. The CPU operating time of the instrumented browser is more than the origin browser because the instrumented browser has 19.08 ~ 43.76% overhead of execution time. However, it does not cause more CPU loading because the tasks of extended time (the three parts which are introduced in previous paragraph) needn’t too much computation resource to do. Fig 13(c) depicts the used memory status. From the figure, the used memory of the instrumented browser is more than the origin browser in the final, because the time profiling component stores the time results in the memory. But it avoids storing large time results by changing profiling resolutions and area. Therefore, the extra overhead of used memory can be controlled below 6.53%. The 6.53% is the memory overhead when we turn on all probes. And the used memory of the instrumented browser is less than the origin browser during the execution period because the actions of allocation and free memory will trigger the VM to do garbage collection. Hence, the used memory will down when the garbage collection recycle the memory in the execution period.

Experiment of time accuracy

In Chapter 4, we showed that 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 at runtime that will be included in the time results of RMP; therefore, we need to deduct the extra time for accuracy of time results.



```
The test case
------------------------
System.getTimeMillis(…)
   for (i=0;i<10000;i++)
      caller = getCaller()
System.getTimeMillis(…)
```

(a) The test case of getCaller().

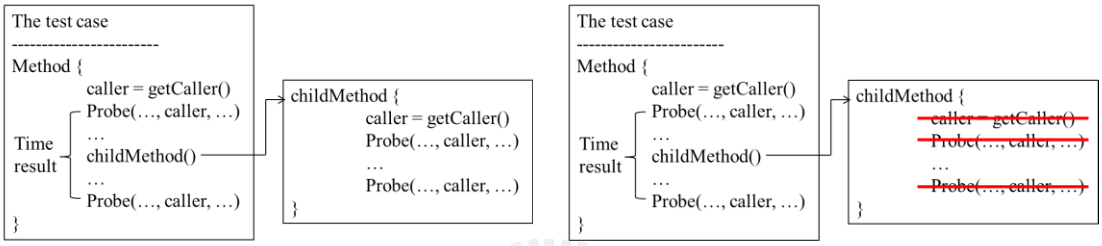
```
The test case
------------------------
System.getTimeMillis(…)
   for (i=0;i<10000;i++)
      Probe(…, caller, …)
System.getTimeMillis(…)
```

(b) The test case of Probe().

Figure 14. The overhead experiment of instrumented methods.

Figure 14 shows the experiments which measure the execution time of getCaller()

and Probe(). The getCaller() and Probe() are executed 10000 times in a loop, and the time information is recorded by instrument System.getTimeMillis() at the head and tail of the loop to get the average execution time of them. The getCaller() spends 0.382 (ms) for one round, which is eight times faster than 3 (ms) for getStack(), and the Probe() spends 0.077 (ms) for one round. Therefore, we can suppose a probe takes extra 0.54 (ms) execution time because a probe includes one getCaller() and two Probe().



(a) Child method is instrumented. (b) Child method without instrumented.

Figure 15. Measure the execution time of a probe.

To evaluate the extra execution time of a probe, we do another experiment as shown in Figure 15. We select a method which has been instrumented and it includes a child method which has been instrumented too as shown in Fig 15(a). Figure 15(b) shows another case, we remove the instrumentation lines (a probe) of the child method. Then we get the time results from each of them and minus the time results to know the extra execution time of a probe is 0.488 (ms). It is close to our supposition. Therefore, we deduct this time value in the time results of RMP by the number of probes in child methods. However, 0.488 (ms) is the time of an application probe, but the extra execution time of a framework probe is different. The framework probes have one more stage in the time profiling component to check whether the caller is the target application for avoiding recording time results which called by other applications. Therefore, we deduct 0.631 (ms), which is got from the same experiment.

Unfortunately, some framework probes are called by some missed instrumented framework methods because the ctags cannot scan out the inner class of Java language (ex. call back function of listener) and our automatic scripts cannot scan out a case of called framework methods. For example, CookieSyncManager.getInstance().stopSync(), we can know the code of getInstance() is written in CookieSyncManager.java directly, but the stopSync() is written in the class which is returned by getInstance(). We cannot instrument the probe into stopSync() because we cannot know what class will be returned by getInstance() in instrumentation ranges analyzer. The information of return value needs other resources to help (ex. Software development kit (SDK)). The log events of framework probes which called by missed instrumented framework methods will be ignored in the time profiling component because the name of the missed instrumented framework methods are not recorded in the caller filter list. However, the time profiling component will spend time to check these log events, and the check time is included in the time results of RMP. Therefore, we select a framework method based on the same scenario in Figure 15 to do the same experiment, but the log event of the child method will be ignored in the time profiling component by configuration settings, not remove the probe. Then we know the extra execution time of the ignored framework methods is 0.583 (ms) and we deduct it in the time results of RMP by the information of ignored times got from the time profiling component.

Table 3. The time error rate of the RMP and debug class.

	onPause	addObserver	resetLockcon	buildTitleUrl
System.nanotime()	14.518 (ms)	1.221 (ms)	1.312 (ms)	1.068 (ms)
RMP	15.562 (ms)	1.167 (ms)	1.282 (ms)	1.177 (ms)
Error rate of RMP	7.19%	0.04 %	2.29 %	10.21%
Debug class	8.25 (ms)	2.417 (ms)	2.099 (ms)	4.061 (ms)

Error rate of debug class	43.17 %	97.95 %	59.98 %	280.24%

In table 3, we select some methods whose execution time is stable to evaluate the error rate of time results. The reference time is got from the API, System.nanotime(). We instrument this API into the entry point and exit point of methods to get the execution time of methods in nanosecond. And according with the above mentioned, the time results of RMP are deducted by some time values to be the final results. The table shows that the error rate of the RMP is below 10.21%.

Compare overhead and time accuracy with debug class + Traceview

The debug class with Traceview is a default time profiling tool of Android official and work on most Android product devices. Therefore, we select this tool to be compared with the RMP on our DUT. Traceview is a GUI tool which can analyze logs recorded from the debug class and show the execution time from process level to method level. The debug class provides the API of start point and end point for users who can instrument them into the application source code to profile all methods of an application, including application methods, framework methods and core library of VM. The debug class collects all time results of methods and stores the results in the memory at run time. It will remove time results from memory to the SD card when the profiling is finished (the API of end point is executed). Therefore, the debug class uses large memory space to store the time results. Traceview can only provide the round times and total execution time of a method and the execution time of each round for a method is the average result in Traceview. However, a method may spend different execution time in different task by different parameters. Therefore, the Traceview cannot help us to analyze the execution time of each round for a method. Table 4 shows the comparisons between debug class + Traceview and RMP.

Table 4. The functional comparisons of debug class + Traceview and RMP.

	Profiling approach	Profiling resolution	Profiling target	Log size	Integral logs problem
Debug class + Traceview	Instrumentation-based approach	Process to method	(1) Application (2) Framework (3) Core library of VM	Large	Yes
RMP	Instrumentation-based approach	Process to loop	(1) Application (2) Framework	Small	No

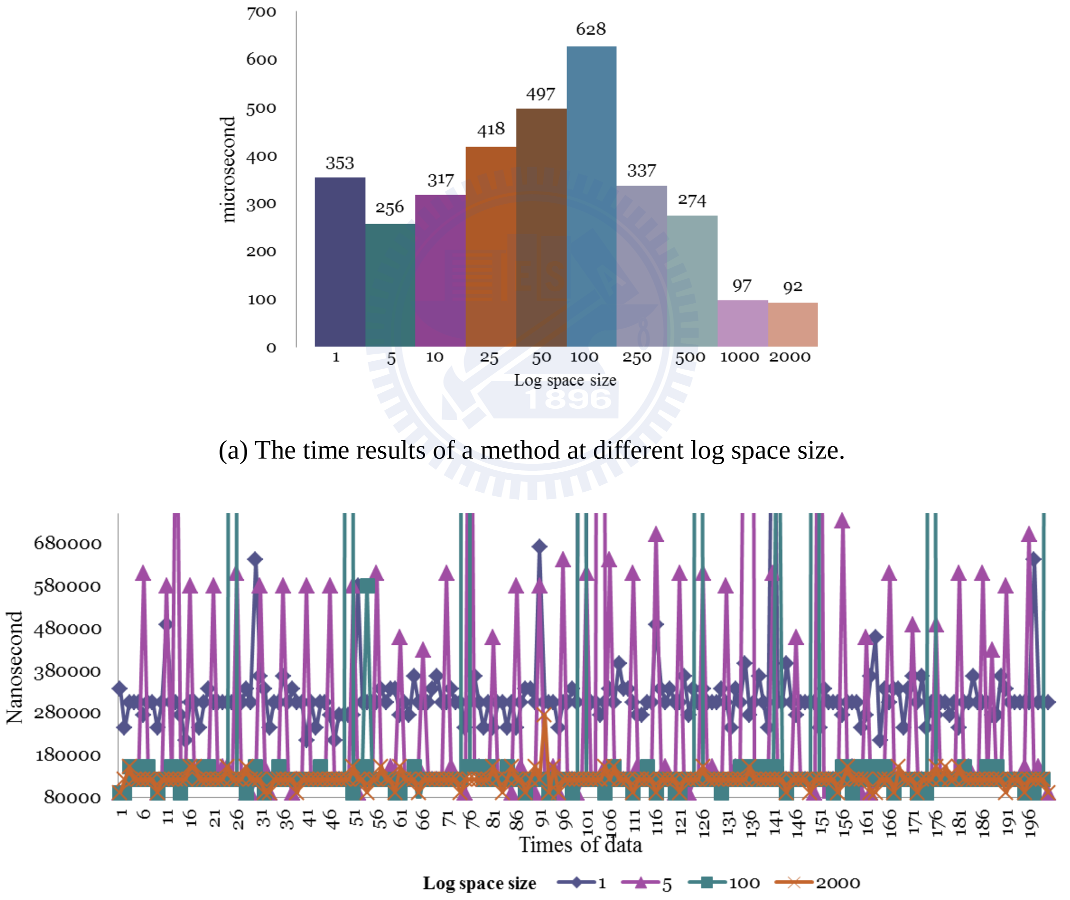
Table 5. The overhead comparisons of debug class and RMP.

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

Table 5 shows the overhead comparisons of debug class and RMP; the debug class extends 39.72% overhead of execution time, 4% CPU overhead and 76.3% memory overhead. The overhead of execution time is caused by the VM which does the initial work and collects the time results of each method in classes. And 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 overhead of debug class and RMP are close. However, the memory overhead of RMP is smaller than that of debug class because the RMP is a reconfigurable multi-resolution profiling solution which can save the log size. Table 3 shows the time error rates of debug class, 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 next executed method, the results are not very accurate. Therefore, the results of debug class cannot exactly help users to find out the bottlenecks.

Experiment of different log space size

In RMP, the log space size is the number of time results can be storage. If the log space size is small, then the time profiling component needs to spend time to remove log results from memory to the ram disk during profiling. The time of the data movement will be included in the time results, which will increase the time result error rate. On the contrary, if the log space size is large, then the time profiling component no need to spend the writing time during profiling. However, the log space size cannot set too large because it will increase the memory overhead. In addition, the memory space of the embedded devices cannot support large log space. Therefore, we need to consider trade-off between memory overhead and time results error rate.



(a) The time results of a method at different log space size.

(b) The time results of a method in each round.

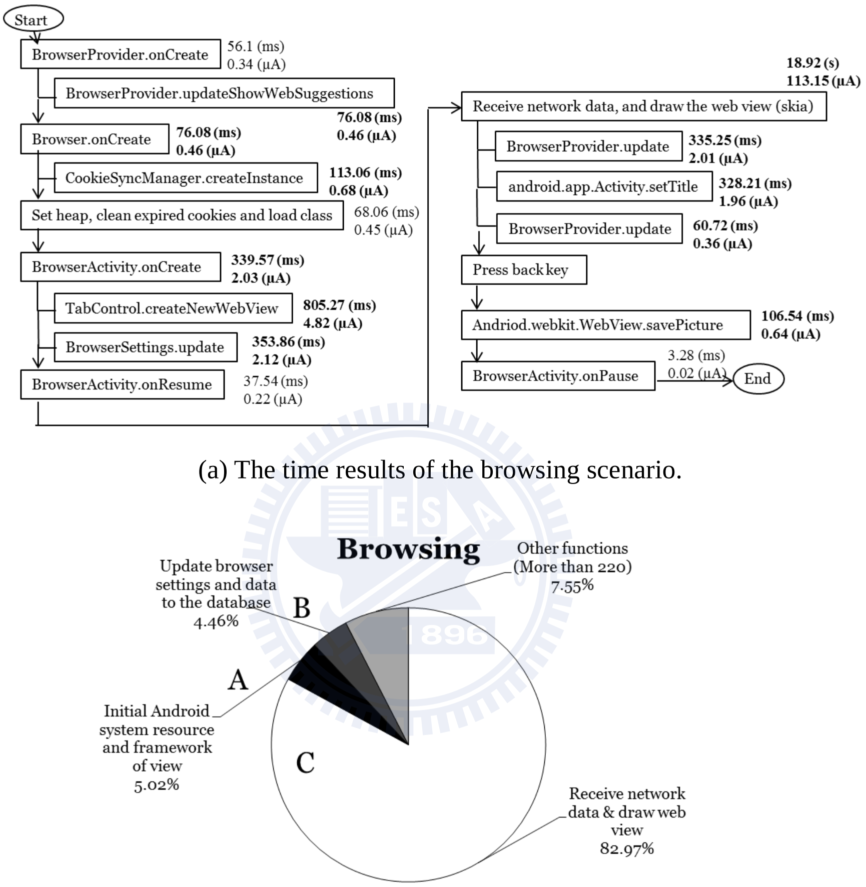
Figure 16. The overhead at different log space size.

In this experiment, we take a music playing scenario with different log space sizes to find the reasonable log space size. In Figure 16, we select a method (MediaPlaybackService.position()), which is executed over 200 rounds in the scenario, to be the experiment target and the range of log space size is 1 ~ 2000 results. When the log space size is one, the time results will be written to the ram disk every time. If the log space size is 2000, the time results will be written to the ram disk when the number of time results is over than 2000. Figure 16(a) depicts the average time results of the method at different log space size. From the figure, the average result for log space size 2000 is more accurate because the number of recorded time results in playing music scenario is less than 2000. In other word, there are no any data movements during profiling. When the log space size is one, the average result is not quite right because it includes large results writing time. Therefore, we increase the log space size from one to five and ten can enhance the result accuracy. However, when we set the log space size to 100, the average result of this size is bad than that of small log space size because the results writing time of 100 log space size is longer. In Figure 16(b), the results writing times of 100 log space size is less than one or five log space size, but the results writing time is longer than that of one or five log space size (The writing time of 100 log space size is out of y-axis in figure). This is because there are 100 results in memory need to be written to ram disk, not only one or five results. Therefore, the time results error rate is minimal if the DUT can provide enough log space to store all profiling results. On the other hand, the log space size needs to consider the available memory space and the number of time results in scenario when the memory space is not enough to store all profiling results, then choice one which can get balance between the writing time and the number of data movement for profiling.

Energy profiling

In RMP, the energy profiling method is the enhanced approach of Battery Use. Its overhead is 3% and the results error rate of process level is 10% which was introduced in [8]. The RMP can provide results of energy consumption at fine-grained resolution by the time proportion method, but we cannot do the evaluation of the results error rate at fine-grained resolution. Because most energy profiling tools (ex. ePro) need the hardware support to do fine-grained profiling, it cannot work on our Android-based product DUT.

5.4 Overall Observations for the Browser



(a) The time results of the browsing scenario.

(b) The bottlenecks of browsing scenario.

Figure 17. The profiling results of browsing scenario.

The Figure 17 shows the profiling results of the browsing scenario by the RMP. The whole profiling time only needs 1 ~ 2 hour, and it is quicker than staged interactive instrumentation approach (SIIP) [20] which need more than one day for the same profiling in our experiment. This is because the RMP instruments all profiling points into source code at first and changes profiling resolutions and area by configuration settings. Users only need to instrument and recompile the profiling application and framework in 40 ~ 45 minutes at first and change the profiling resolutions and area in 5 minutes for each round. On the contrary, the SIIP need to spend time to instrument and remove the profiling points when users change the profiling area and wait the recompile time (35 ~ 40 minutes of each round in AOSP environment) for each round. The average log size of the RMP is 15 KB; it is smaller than the log size of debug class (2.99 MB) 25 times because the RMP can only record the useful logs by setting profiling resolutions and area for each round.

The Figure 17(a) shows the detailed information about execution time and energy consumption for the browsing scenario. It can be modeled to pie chart according to their function. Figure 17(b) shows the pie chart for the browsing scenario. The “A” part takes 5.02% of the browsing time for initialing the Android resource (Ex. Skia graphics library (SGL) [21]) 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 draw web view are implemented in C/C++ library, so we cannot know the detailed time distributions for these actions. But we know the bottleneck is the actions of draw web view by analyzing the library source code. The view of 3D is drawn by OpenGL ES [22] which uses the graphic processing unit (GPU) to do hardware acceleration, but the view of 2D is drawn by SGL which does not use any hardware acceleration to draw the web page. It is slow when the web page is drawn by the embedded CPU. Therefore, we hope the SGL can support hardware acceleration to optimize the bottleneck of browsing in the future version.

Chapter 6. Conclusions and Future Works

This work designs the reconfigurable multi-resolution profiling approach to profile execution time and energy consumption of applications by using limited log space for finding the bottlenecks on the resource-constrained embedded system. The approach was practiced on the off-the-shelf product to examine the bottlenecks of the browsing scenario.

In all evaluation studies, the overhead of our implementation is proven with minor CPU overhead (5%) and memory overhead (below 6.53%). Although the overhead of execution time is 19.08 ~ 43.76%, we can correctly deduct them to get the correct execution time result of methods. The accuracy of execution time results is evaluated with the average error ratios below 10.21% and we further show that RMP results are more accurate than results of debug class 24 times. In addition, the RMP may be faced with a trade-off between memory overhead and accuracy of the time profiling when the memory space of DUT is not enough to store all profiling results, so it's important to choice a reasonable log space size according to the memory space and the amount of profiling results. Then the choose log space size can get balance between the writing time and the number of data movement for profiling.

From the case studies, we observed that the bottleneck of the browsing is the web page drawing which takes a lot portion of total execution time and energy consumption. This is because the graphics library, SGL, uses non-powerful embedded CPU to draw the web page and does not use any hardware acceleration.

Finally, because of the lack of profiling on C/C++ source code for graphics and network library, the profiling of the browsing can only know the entry and exit execution time of the library. We still have to spend extra time to analyze the source code of the library for finding the bottlenecks. Therefore, the future extension of this work includes the profiling of C/C++ source code. It can help us to profile applications which are written in C/C++ language, e.g., the video application on Android. And we want this work can be extended to profile the core library of VM for users who can know more detailed information too. Besides, we want to fix the missed framework instrumentation problem introduced in Chapter 5, such as use the compiler to do instrumentation, and reduce the overhead of extra execution time by enhancing the instrumentation method and the decision approaches of the time profiling component in the future.

Reference

[1] J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, 1999, pp. 2-10.

[2] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 17, pp. 120-126, 1982.

[3] M. Desnoyers and M. Dagenais, "LTTng: Tracing Across Execution Layers, From the Hypervisor to User-Space," in *Proceedings of the Ottawa Linux Symposium*, 2008, pp. 101-105.

[4] "Android Debug." [Online]. Available: http://developer.android.com/reference/android/os/Debug.html.

[5] "Oprofile: A System Profiler for Linux." [Online]. Available: http://oprofile.sourceforge.net/.

[6] L. Adhianto, *et al.*, "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice and Experience*, 2009, vol. 22, pp. 685-701.

[7] "Intel VTune Performance Analyzer." [Online]. Available: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[8] Y. Yun-Chien and L. Ying-Dar, "Calibrating Parameters and Formula for Process-Level Energy Consumption Profiling," Mater thesis, NCTU, 2010.

[9] "Kernel Function Trace." [Online]. Available: http://elinux.org/Kernel_Function_Trace.

[10] "TraceView." [Online]. Available: http://developer.android.com/guide/developing/tools/traceview.html.

[11] T. L. Cignetti, K. Komarov, and C. S. Ellis, "Energy Estimation Tools for the Palm," in *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Boston, Massachusetts, United States, 2000, pp. 96-103.

[12] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir, "Using Complete Machine Simulatiion for Software Power Estimation: The SoftWatt Approach," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, 2002, pp. 141-150.

[13] W. K. Baek, Y. J. Kim, and J. H. Kim, "ePRO: A Tool for Energy and Performance Profiling for Embedded Applications," *Proc. of International SoC Design Conference (ISOCC)*, Seoul, Korea, Oct. 2004, pp. 372-375.

[14] T. Do, S. Rawshdeh, and W. Shi, "pTop: A Process-Level Power Profiling

Tool," presented at the Workshop on Power Aware Computing and Systems, 2009.

[15] K. S. Banerjee and E. Agu, "PowerSpy: Fine-grained Software Energy Profiling for Mobile Devices," in *Proc. of IEEE WirelessCom*, 2005, pp. 1136-1141.

[16] "Jindent." [Online]. Available: http://www.jindent.com/.

[17] "Exuberant Ctags." [Online]. Available: http://ctags.sourceforge.net/.

[18] L. Batyuk*, et al.*, "Developing and Benchmarking Native Linux Applications on Android," in *MobileWireless Middleware, Operating Systems, and Applications*. vol. 7, J.-M. Bonnin, C. Giannelli, and T. Magedanz, Eds., ed: Springer Berlin Heidelberg, 2009, pp. 381-392.

[29] "JNI Enhancements Introduced in Version 1.2 of the Java 2$^{\text{TM}}$ SDK." [Online]. Available: http://download.oracle.com/javase/1.4.2/docs/guide/jni/jni-12.html.

[20] D. Tzu-Hsiung and L. Ying-Dar, "Booting, Browsing and Streaming Time Profiling and Bottleneck Analysis on Android-Based Systems," Mater thesis, NCTU, 2010.

[21] "Skia graphics library." [Online]. Available: http://code.google.com/p/skia/.

[22] "OpenGL ES." [Online]. Available: http://www.khronos.org/opengles/.