Palmist: A Tool to log Palm System Activity

Raghunath Gannamaraju and Surendar Chandra
Department of Computer Science, University of Georgia, Athens, GA 30602-7404
{*raghuna,surendar*}*@cs.uga.edu*

Abstract

In this paper we describe a Palm system call logging tool called Palmist. Palmist allows the practitioner to selectively collect statistics such as the system call invoked, application that invoked the system call, the time of the call and the call arguments. The logging mechanism adds a latency of about 10 msec per call to collect the log. On an average, the system uses about 20 bytes of memory on the PDA to store the log record. The mechanism has limitations in collecting logs for system calls that are needed by the collection mechanism itself. Our logging mechanism works for about 80% (707 of 881) of Palm OS 3.5 system calls. Our system can be utilized by system developers to customize their application behavior to optimize system parameters such as energy consumption, ease of use etc.

1. Introduction

There is an explosion in the number and variety of handheld devices. Falling prices and the increasing availability of newer applications is driving the popularity of hand held devices. Hand held devices are constrained in their usability by the available network, battery capacity and display characteristics. The typical client usage pattern has a profound impact on the usability and the efficacy of system management policies. For example, the access pattern of the user significantly affects the battery energy consumption. Depending on the idle duration, the Palm device consumes more energy to shut down and power up than to remain idle [2]. A policy that conserves energy by shutting down the unit after a fixed idle period would not be effective if the user resumes work immediately, forcing a powerup. Policies that can change the computation to leverage these device characteristics can be expected to prolong battery life. Understanding the actual user behavior is fundamental to designing these newer policies.

Representative client usage logs are invaluable in answering realistic questions about the system behavior of PDAs. However, little is known of the actual usage behavior of hand held devices. Earlier work [3, 2] used oscilloscopes to capture system activity. Such probes restrict the ability to collect mobile user activity. Also, publicly available Palm OS Emulator (POSE) [4] is widely used for debugging and profiling Palm applications. These emulation programs typically run on desktop computers. Hence, their applicability to profile and capture mobile users' behavior is limited. There is a need to instrument the actual devices to capture the mobile user behavior.

The primary goal of this work is to develop such tools to capture the system behavior of PDA devices. The logs collected can be used by practitioners to evaluate the performance of their system design. For example, operating system developers for the PDA platform can analyze the efficacy of their energy saving features using a typical user. Application designers can also improve the general usability by improving the accessibility of popular features.

Palm OS [11] based PDAs account for 87% of the hand devices sold [9]. Hence, we focus our attention on Palm devices. We describe Palmist, a tool to log system calls for the Palm device. One of the design goals in developing Palmist was to capture enough system activity details to replay them in log play back tools. The Palms were instrumented using the *hack* [5] mechanism to collect statistics such as system call invoked, application that invoked the system call, the time of the call and the call arguments. The system allows the practitioner to selectively log a subset of interesting system calls. The collected information can be periodically *sync*ed to a desktop computer for further processing. These instrumented Palms can be distributed to *typical target users* and their usage patterns collected over a period of time. The trends in the users' access patterns can be analyzed.

One of the challenges in designing Palmist was to constrain the overhead of the additional instrumentation. The overhead introduced by Palmist depends on the amount of system call activity logged. We show that our logging mechanism adds an acceptable amount of overhead in collecting the system call events. On an average, the log records consume about 20 bytes of storage on the Palm device. The Palmist trap handlers add an additional latency of about 10 msec per system call logged. Depending on the application scenario, we show that in the worst case, a fully instrumented Palm can run up to two orders of magnitude slower. However, disabling a few popular calls and using a faster PDA adds an extra 50% overhead for interactive tasks. The mechanism also has limitations in collecting logs for system calls that are needed by the collection mechanism itself. Our logging mechanism works for about 80% (707 of 881) of the Palm OS system calls.

The remainder of this paper is organized as follows: in Section 2., we begin by laying the background for our work by briefly describing the Palm OS features used in developing our system. We describe the Palmist tool in detail in Section 3. Experimental setup to measure the logging overhead and the evaluation methodologies used in our study are described in Section 4. Section 5., places our work in context to other related work in the field with conclusions in Section 6.

2. Background

In this section, we briefly describe several features of Palm OS that were exploited in developing Palmist. In particular, we describe how the Palm OS system handlers can be replaced with our own trap handlers, how the *feature manager* can be utilized to provide global memory regions and how Palm stores permanent data in databases.

2.1. Trap handling

The Palm OS supports a trap mechanism to service system events and system calls, through a system-wide trap handler vector (illustrated in Figure 1(a)). Each system call has a 16-bit Trap ID value associated with it. By default, when an event occurs or a system call is executed, a lookup is done in trap handler vector and the appropriate Palm OS system call trap handler is executed. Based on their functionality, the system calls can be broadly categorized into various APIs such as Form API, Database API, Resource Manager API etc. Palm OS 3.5 supports a total of 881 system calls.

Palm OS allows users to modify the default system behavior by installing their own trap handlers. These user defined trap handlers are generally known as *hacks*. These *hacks* are in the form of Palm resource files (.prc) of type *HACK* and contain the user-defined handler code (as a code resource) along with the trap ID of the system call being handled. Together, the file type and the trap ID are used to get the location of the user-defined handler. If a hack is activated, the user-defined code is executed in lieu of the original Palm OS provided trap handler.

2.2. Feature Manager

Palm OS application's global data is not generally available outside the application. *Feature* memory provides access to data that persists between invocations of an application. *Feature* memory is allocated from the storage heap. The values stored in *feature* memory persist until the device is reset or until the memory is explicitly freed. Applications can share data by publishing a *feature* that contains a pointer to the shared data.

2.3. Palm Databases

Palm databases function similar to a file storage abstraction. All Palm OS data, including RAM and ROM applications, user data and application preferences are stored in databases. The Palm database is composed of unstructured records. There is no inherent limit on database size (as long as it fits into memory). However, records within a database are restricted to about 64 KB.

3. System Architecture

3.1. Objectives

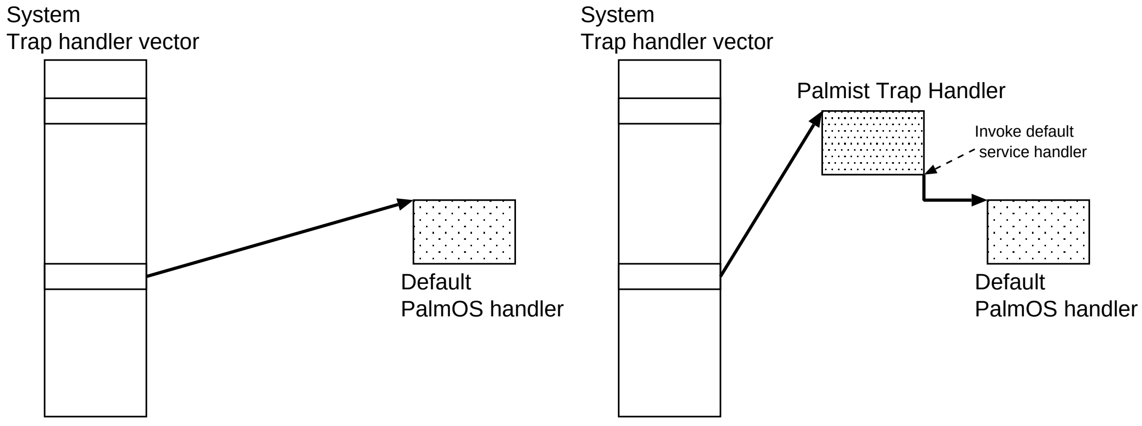
The primary goal in designing the Palmist system was to develop the functionality required to capture information about system events. We wanted to capture enough information so that play-back tools can be developed to recreate a particular user interaction. Palmist instruments the system calls to collect the following per-call information:

- system call invoked,
- application that invoked the system call,
- the start time of the call since last reset (in 10 msec increments)
- the system call arguments. Pointers are dereferenced up to one level of indirection.

3.2. Palmist trap handler installation mechanism

We utilized the *hack* mechanism to develop our logging tool. We developed a unique system call specific *hack* to collect these logs. Tools such as *hackmaster* [5] can manage the activation and deactivation of the *hack* extensions to the Palm OS system software. However, there are limitations on the number of *hacks* handled by hackmaster. Since Palm OS 3.5 provides 881 system calls, we developed our own strategy to install these user defined trap handlers. Throughout the rest of this paper, we refer to these *hacks* as Palmist trap handlers.

We utilized the Palm OS system calls *SysGetTrapAddress* and *SysSetTrapAddress* to develop our system. For a given trap ID, *SysGetTrapAddress* returns the address of the original system trap handler, while *SysSetTrapAddress* can be used to change the system trap handler's address. Palmist trap handlers maintain the original functionality of the system by storing the original trap handler (returned from *SysGetTrapAddress*) before updating the system trap vector us-



(a) Palm OS system handler

(b) Palmist patch to system traps

Figure 1: Hack mechanism to trap system calls

ing *SysSetTrapAddress*. We utilize the *feature manager* facility *FtrSet* to store the original system trap handler address. After the log collection, the original system call is invoked from within our logging function (Figure 1(b)).

3.3. Palmist trap handlers

An unique Palmist trap handler is installed for each system call that needs to be logged. These Palmist trap handlers have the same function prototype as that of the original system call. The function parameters to the original system call are passed on to the Palmist trap handler.

In order to reduce the overall code size for all the unique Palmist trap handlers, we utilize a common logging library function from these trap handlers. In the Palm OS, library functions need to be loaded first before they can be invoked. In order to reduce the logging overhead of loading and opening the library for every system call (quantitatively measured in Section 3.3.1.), we initially loaded and opened the library once while installing the Palmist trap handlers. Subsequent invocations directly call this (already loaded and opened) library function. Throughout the data logging, this library is never closed.

Each trap handler passes the system call's trap ID as well as a packed argument buffer to the common logging library function. While packing the arguments, pointers are dereferenced up to one level for *string* and *struct* data structures. The distribution of the number of parameters to system calls are tabulated in Table 1. Of the 707 system calls that can be instrumented with Palmist, 276 system calls do not return a value.

The common logging library function logs the system call parameters along with the current system call start time (in 10 msec ticks since last reset) and the application ID of the active application. Application ID is the database ID of the current application's resource database. Application ID is unique between Palm resets. The application names associated with each application ID can be accessed directly from the 'tAIN' resource during Palmist initialization.

No. of parameters	No.of system calls
0	89
1	211
2	183
3	110
4	57
5	38
6	9
7	6
8	1
9	4
10	0
11	2
12	0
13	1

Table 1: Distribution of the number of parameters to Palm OS 3.5 system calls logged with Palmist

The common logging library function stores the system call log information in a Palm database. This storage log database is initially created during the installation process. In order to reduce the latency in opening this log database, we open the database once during this initialization and store the database handle of this database as a *feature*. Subsequently, Palmist reuses this database handle to open the database.

The common logging library function appends new log information into this storage database. Each Palm record has an associated meta-data of 8 bytes. In order to minimize the storage requirements of this per-record meta-data, we pack (by appending to the last record) the logs into records that can be up to 64 KB (Palm OS limitation). New log data are always appended to the last record; the position at which the next data item is to be written is stored in the first two bytes of the last record in the database. When the last record fills up, a new record is created.

Such logging mechanisms introduce additional data collection overhead. As we will note later in Section 3.3.2., system restrictions allow us to only capture about 80% (707 of 881) of the system calls using Palmist trap handlers.

3.3.1. Palmist trap handler optimizations

System call logging introduces additional overhead to the normal operation of the Palm device. In order to identify the expensive operations in our log collection strategy, we profiled the entire Palmist code. We measured the difference in system time before and after a particular operation. On the Palm device, the system clock can be measured with a granularity of 10 msec ticks. We noted that library load and open calls as well as *DmOpenDatabaseByTypeCreator* are expensive.

The library load and open calls were optimized by initially preloading these functions during the Palmist installation. Further calls reuse these preloaded functions to reduce the logging overhead. Also, with *DmOpenDatabaseByTypeCreator*, the system searches for the particular database and hence can take upto 6.2 msec per call. We replaced the *DmOpenDatabaseByTypeCreator* with *DmOpenDatabase*. *DmOpenDatabase* needs the database ID instead of the database name to open the particular Palm database. This database ID is unique between system resets. We locate this ID during the Palmist trap handler installation and store the value as a *feature*. This database ID is queried from the *feature manager* before opening the database using *DmOpenDatabase*. The *DmOpenDatabase*, *FtrGet* combination consumes 3.7 msec; a savings of 2.5 msec per Palmist trap handler invocation.

3.3.2. Log collection limitations

Our system call collection mechanism has inherent limitations in collecting the activity of certain system calls. System calls that are needed within the Palmist trap handler code themselves cannot be logged because of recursion. Several system calls (low level memory functions) are also indirectly referenced by other system calls used by Palmist. For example, we utilize the system call *DmCloseDatabase* within the common logging library. Instrumenting *DmCloseDatabase* causes stack overflow; resulting in a system crash.

Several system calls are called exclusively during the HotSync process. Since we disable logging during the HotSync operation (as described in Section 3.4.), these calls cannot be logged.

Also, Palm applications have access to limited stack space. Instrumenting system calls necessitate additional stack space. Application developers can increase this default stack space. Since we do not modify the actual applications directly, some system calls which call other system calls repeatedly cause a system stack overflow. Hence, logging was turned off for such system calls.

Some of the system calls that are not logged because of these limitations include *MemHandleNew*, *MemHandleLock*, *MemHandleUnlock*, *MemHandleFree*, *MemCardInfo*, *TimGetTicks*, *SysCurAppDatabase*, *DmOpenDatabase*, *DmNumRecords*, *DmNewRecord*, *DmWrite*, *DmReleaseRecord*, *DmGetRecord* and *DmCloseDatabase*. We are currently exploring ways to log some of these system calls using alternative techniques.

3.4. Downloading Palm's activity log onto the desktop

We utilized the HotSync process to download the logs from the Palm to the desktop [10]. We developed a conduit to download the system call log data collected by Palmist. We utilized the *SyncManager* API to develop this conduit. During the HotSync process, the log storage database is opened to upload data to the desktop. The conduit downloads the data record by record from the Palm storage database. Since the database is already opened by the conduit process, it is not possible to collect system logs while HotSyncing the logs. Palmist traps check for this condition and disable logging as long as the system is performing a HotSync of the log database. The collected statistics are written into a text log file on the host desktop. The records are then deleted from the Palm database to free up storage resources.

3.5. Experimental setup

Palmist was developed to collect system call information for Palm OS 3.5 [1]. Palm OS 3.5 supports a total of 881 system calls. As we will describe in Section 3.3.2., Palmist can only collect access logs for 707 system calls. In general, Palmist can be extended to collect the logs for newer calls introduced in Palm OS 4.0. We developed the Palm conduits using Conduit SDK 4.0.2. We used Code warrior R6 on a Windows NT desktop to develop Palmist. We've tested Palmist on IIIxc, IIIxe and m505 Palm computing devices.

4. Performance Results

In the last section, we described the Palmist tool and how it collects log information about system call activity. In this section, we present our results and experiences in using the tool as well as the overhead in collecting such usage logs.

We describe the memory and log collection latency overheads in using Palmist. We show the measured overhead for targeted micro-benchmarks. We also outline a usage scenario to highlight how Palmist can be utilized to identify performance bottle-necks in Palm applications. We analyzed the system calls invoked for a typical session using the Palmist tool.

4.1. Log collection overhead

First we explore the overhead for collecting usage logs. We describe the storage overhead in collecting logs. Then we describe the HotSync overhead in downloading the logs. Next we utilize our own microbenchmarks to experimentally measure the per system call latency as well as cumulative slowdown in collecting the system activity.

4.1.1. Log collection storage overhead

Palmist's installation on Palm entails memory overhead, both for the code of Palmist trap handlers as well as the logs that are collected using Palmist system. Each Palmist trap handler code occupies about 400 to 700 bytes of memory (depending on the number of arguments to the particular system call) along with another 2.53 KB for the common logging library function. The Palmist system code consumed about 700 KB for all the 707 Palmist trap handlers. It should be noted that a typical Palm device offers about 7.7 MB of free storage (before installing additional software).

The system call information itself is logged in a log database on the Palm device. Such logging consumes the limited amount of memory available in a Palm device. Each log record consumes at least 12 bytes of storage space to log the Trap ID, system time (in 10 msec ticks since the system reset) and the application ID. The system call parameters consume additional storage space. The size of the entire log is limited by the amount of memory available in the system. Each record in the Palm database has an additional overhead of 8 bytes to store meta-data. Hence, we pack multiple logs into a single Palm record to amortize this storage cost.

When all the available memory is consumed by the logging database, the system crashes for lack of memory for essential system operations. Hence, Palmist disables logging when the available memory is lower than a configurable threshold. For our experiments, we chose a threshold of 512 KB. Palmist checks to see if the available memory is above this threshold before storing any logs. Once the logs fill-up the available memory, the user has to HotSync the logs to the desktop before further logging is possible.

4.1.2. Hotsync latency

Uploading the logs from the Palm to the desktop adds additional latency during the HotSync process. For example, HotSyncing a 1064 KB log from a Palm IIIxe to a desktop with dual 933MHz Pentium III and 512 MB memory using a serial cable connected at 115 kbps adds about 2 minutes to the HotSync process.

Operation	System call	Time (in msec)
	OriginalCall	0.47
Log record	TimGetTicks	0.03
	CurrAppDatabase	0.15
Memory functions	MemHandleNew	0.66
	MemHandleFree	0.57
	MemHandleLock/ MemHandleUnlock	0.85
	MemCardInfo	1.05
Storage functions	FtrGet	0.39
	DmOpenDatabase	3.30
	DmWrite	2.82
	DmGetRecord	0.47
	DmReleaseRecord	0.41

Table 2: Relative overhead for Palmist internal system calls

4.1.3. Log collection latency

The Palmist trap handlers add additional logging overhead to Palm system calls. In this section, we measure the per system call overhead as well as the typical slowdown for Palmist instrumented systems.

First we identify the latency introduced by Palmist for instrumented system calls. We chose five popular system calls that were invoked during the usage sessions described in Section 4.2. We measured the time for 1000 invocations of the system calls with and without the Palmist trap handlers and tabulated the results in Table 5. From Table 5, we note that even with the optimizations outlined in Section 3.3.1., the per call log overhead is quite high; upto two orders of magnitude slower. The relative overhead depends on the actual system call; Palmist overhead for simple system calls that return a pointer have an higher relative overhead compared to inherently expensive system calls (such as *DmOpenDatabase*). We measured the overhead of the various system calls used within Palmist for instrumenting *DmWrite* (a relatively expensive system call) and tabulated the results in Table 2. From Table 2, we note that the log storage functions and memory are expensive; consuming as much as 1500% and 665% of the original call, respectively.

However, the actual slowdown experienced by a Palm user depends on the applications' system call vs application code mix. We expect typical Palm applications to be highly interactive with a low ratio of system call time to application time. In order to measure the actual impact of Palmist on a Palm user, we measured the time taken to perform typical operations; add a new contact using the *Address Book* application, perform a simple calculation using the *Calculator* application and add a new appointment into the *Date Book*. Note that the user interaction period depends on the users'

Operation	# Palmist handlers	Time w/ Palm IIIxe (in secs)	Time w/ Palm m505 (in secs)
Address Book (30 sec w/o Palmist)	All	240	110
	w/o top 3	200	85
	w/o top 7	180	80
	w/o top 10	130	67
Calculator (10 sec w/o Palmist)	All	65	28
	w/o top 4	45	21
	w/o top 5	45	20
	w/o top 10	32	18
Date book (20 sec w/o Palmist)	All	118	45
	w/o top 5	75	36
	w/o top 10	60	30

Table 3: Time taken for typical operations with Palmist

familiarity with the Palm device; we expect a user who is less familiar with graffiti input mechanism to take longer to perform these activities. We repeated the experiments by selectively disabling the most popular system calls utilized for these operations. We also repeated these experiments by instrumenting a newer Palm m505 (which uses a 33 MHz processor as compared to a 20 MHz processor used in the Palm IIIxe). It should be noted that m505's run Palm OS 4.0. However, Palm OS 4.0 is upward compatible and can still run a Palm OS 3.5 based Palmist. Our goal was to investigate the usefulness of a faster processor in reducing the logging overhead. For these interactive tasks, the faster processor would not be noticeable to the typical user. We tabulate the times for the various interactive operations in Table 3. From Table 3, we observe a slowdown of upto 8 times (*Address Book*) while using a Palm IIIxe. A faster m505 reduces the Palmist trap handler overhead for the same operation to about 3.5 times. Disabling the Palmist trap handlers for popular system calls shows improvement; disabling the ten most popular system calls along with utilizing a faster PDA reduces the overhead to about 150% of the original operation (*Date Book*).

These results show that, even for an experienced Palm user, a combination of a faster PDA and selective disabling of logging of popular system calls can aid in limiting the logging latency to acceptable levels.

4.2. Palmist Usage Examples

In the last section (Section 4.1.) we analyzed the overhead in using Palmist. In this section, we explore two usage scenarios to highlight the applicability of Palmist in collecting the Palm system activity. Depending on developers' needs, Palmist can be tailored to collect the statistics for a few interesting system calls (rather than all calls possible with Palmist). First we utilize Palmist to analyze a representative user interaction with a Palm device. We also use Palmist to analyze the Pinemark benchmark program.

Type of Test	Benchmark w/o Palmist (in Pinemarks)	Benchmark w/ Palmist (in Pinemarks)
Text test	831.5	88634.6
Database test	151.0	2468.1
Math test	58.5	429

Table 4: Pinemark Benchmarks

System call	Latency w/o Palmist (in 10 msec ticks)	Latency w/ Palmist (in 10 msec ticks)
WinClipRectangle	7	921
WinSetDrawWindow	11	1169
RctCopyRectangle	3	921
WinGetDrawWindow	3	915
FntSetFont	6	1171

Table 5: Palmist overhead per system call

For a representative user interaction we performed the following operations on an instrumented Palm; launch the *Address Book* application, added a contact information (Last name: Foo, First name: Bar, Company: FooBar, Work: 1234567) and then removed a contact address (Accessories, Palm Computing contact). Next we launched the *Calculator* application and calculated $500000 \div 12$ before launching the *Date Book* application and making an appointment ("Meeting" at 10:00) for the next day.

The collected logs were then uploaded to the desktop and analyzed. During our "representative" user session, a total of 68608 system calls were invoked. The total amount of data collected from this representative user session was 1.34 MB. We plot the invocation frequency of the system calls as a cumulative distribution in Figure 2(a). From Figure 2(a) we note that not all system calls were invoked during this user session; only 296 system calls (of 707 instrumented calls) were invoked. We also note that a few system calls were quite popular; 10 calls were invoked for 50% of the time. We tabulate these ten most popular system calls in Figure 2(b). We also plot the frequency of argument sizes to system calls in Figure 2(c). Most of the system calls have no arguments. On average, the argument buffer consumes 8 bytes of storage.

Next we utilized Palmist to analyze Pinemark [12], a publicly available Palm benchmark. Pinemark measures the performance in pinemark units which roughly indicates the amount of time taken to perform certain categories of operations. Pinemark supports three benchmark categories. The text benchmark tests text display functionality, the database benchmark tests the performance of database operations such as open, read, write and close while the math benchmark analyzes the arithmetic performance.

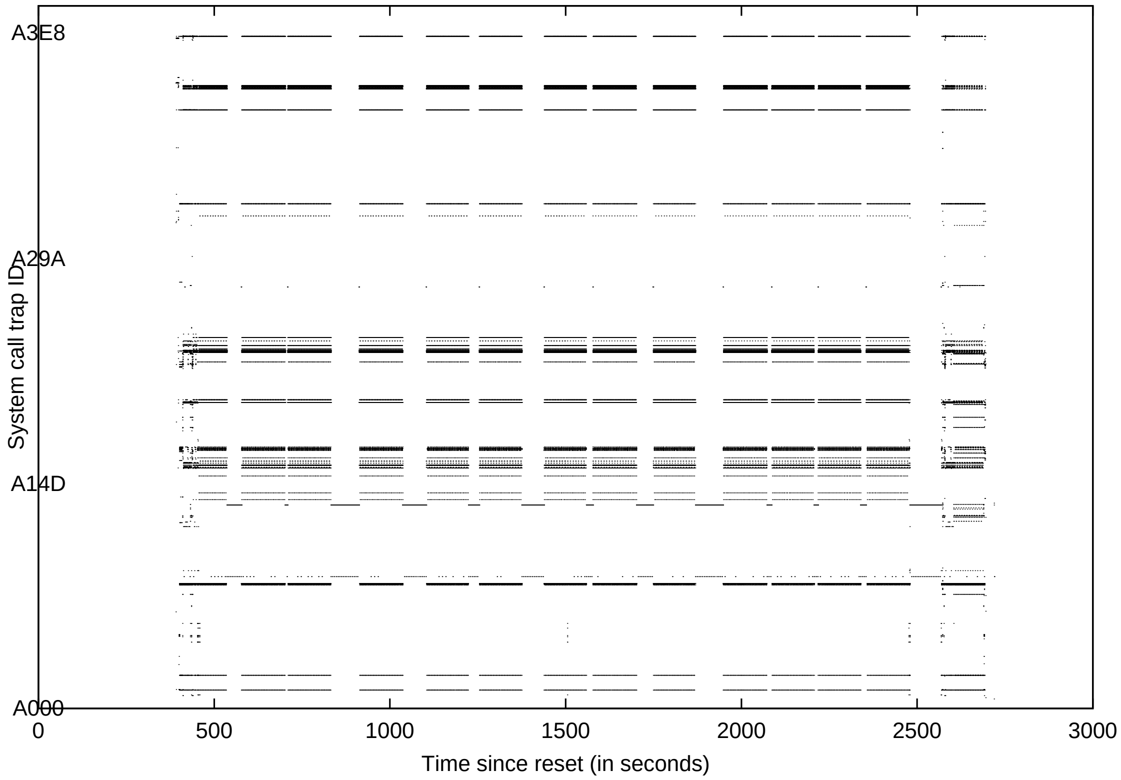


Figure 4: System calls invoked with time (Text Pinemark benchmark)

We benchmarked the Palm with and without installing Palmist trap handlers. The results (averaged over 3 trials) were tabulated in Table 4. From Table 4, we note that the Text benchmarks are significantly affected by the Palmist trap handlers. Math and Database benchmarks showed a moderate slowdown. Most of the system calls exercised by the Text system call were instrumented with Palmist trap handlers. Since the Text benchmark repeatedly exercises these instrumented system calls, the slow down is pronounced. On the other hand, most of the database related systems calls are not logged using Palmist (as discussed in Section 3.3.2.). Hence, the Database test shows minimal slowdown.

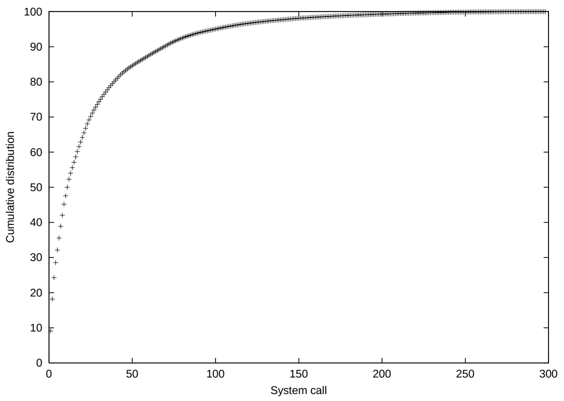
During the text benchmark, the system collected 3.05 MB of trace logs. The Palmist system traced 185,361 systems calls; 177,589 of which were directly invoked by the benchmark. From Figure 3(a), we note that the ten most popular system calls account for over 65% of the system calls. The ten most popular system calls for the Text benchmark are listed in Figure 3(b). We observe that *SysLibTblEntry*, *BmpGetBits*, *BmpColortableSize*, *WinClipRectangle* and *RctCopyRectangle* are among most frequently called system calls both in our representative usage session as well as the Text test of Pinemark benchmarking program. The argument buffer size (shown in Figure 3(c)) is similar to our "representative" session.

Next we utilize Palmist to show how a developer might use Palmist to understand the performance bottlenecks within the Text benchmark. We plot the system calls invoked (trap ID) with respect to the time of invocation in Figure 4. The developer will notice the frequent breaks in system activity (for example, around 1050 secs). They can further explore this particular time interval in greater detail.

In this section, we described some of the usage scenarios for the Palmist tool. We used the tool to identify popular system calls and the system calling sequence for a mobile user. These calling patterns could assist the Palm system developers to selectively optimize these "typical" and popular calling sequences. Techniques such as [7] can be employed to optimize the Palm code. The application developers can also utilize this information to better customize their applications. They can tune these optimizations based on actual mobile user behavior, rather than using a simulated session.

5. Related Work

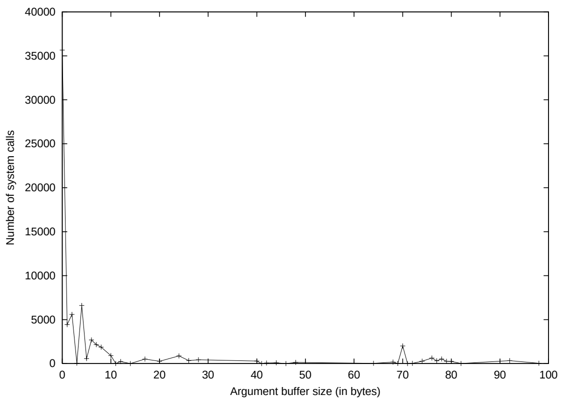
A number of earlier efforts have explored various techniques to capture system behavior. Tools such as SimplePower [13] are cycle accurate execution driven simulators. Tools such as POSE [4] emulate the systems and can be uti-



(a) Cumulative distribution of the number of times a system call was invoked

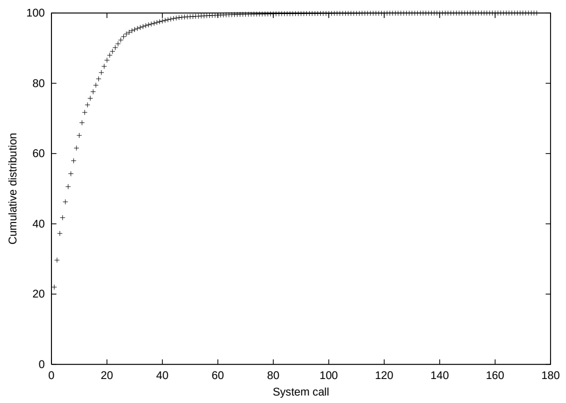
System call	Count
FntSetFont	1662
DmDatabaseProtect	2138
WinRGBToIndex	2148
RctCopyRectangle	2309
BmpGetColortable	2336
WinClipRectangle	2474
WinSetDrawWindow	2944
SysLibTblEntry	4172
BmpGetBits	6233
BmpColortableSize	6236

(b) Occurrence frequency of ten most popular system calls



(c) Distribution of system call argument size (in bytes)

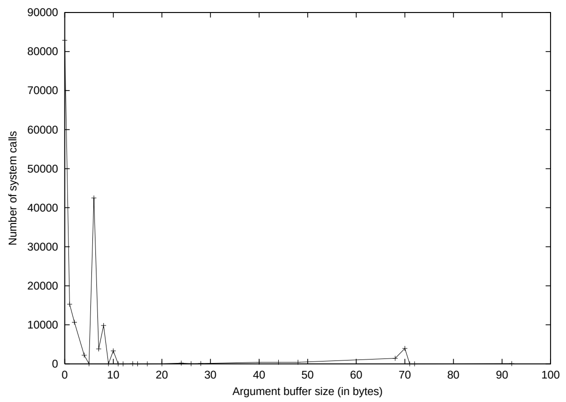
Figure 2: "Representative" session



(a) Cumulative distribution of the number of times a system call was invoked

System call	Count
WinPushDrawState	6408
WinPopDrawState	6408
WinGetClip	6416
WinSetClip	6543
WinResetClip	6553
SysLibTblEntry*	7671
BmpGetBits*	7992
BmpColortableSize*	8995
WinClipRectangle*	13476
RctCopyRectangle*	13633
EvtEnqueueKey	39056

(b) Occurrence frequency of eleven most popular system calls



(c) Distribution of system call argument size (in bytes)

Figure 3: Text Pinemark benchmark (Note: Calls marked with a * were also popular in the "representative" session)

lized to profile a typical user session. Other research efforts such as [3, 2, 8] have used hardware probes to capture system usage information. However, such techniques have limited applicability in a mobile scenario. Mobile users can be expected to use their devices differently while on the road compared to a controlled setting near tethered and bulky simulators and capture devices.

Newman et al. [8] outline an approach very similar to our Palmist approach. However, they abandoned this approach (because of time constraints) for a microbenchmark based technique to measure the power consumption characteristics of the Palm device.

The VTrace logging tool [6] for windows clients is close in spirit to Palmist. In general, the system development and profiling tool support for mobile PDAs lag their desktop computer counterparts. Palmist system logging has higher data collection overhead as compared to VTrace.

6. Conclusions

In this paper, we described a Palm system activity logging tool called Palmist. Palmist allows the practitioner to selectively collect statistics such as the system call invoked, application that invoked the system call, the time of the call and the call arguments. We show that the overhead introduced by Palmist depends on the amount of system call activity logged. We show that our logging mechanism adds an acceptable amount of overhead in collecting the system call events. On an average, the log records consume about 20 bytes of storage on the Palm device. The Palmist trap handlers add an additional latency of about 10 msec per system call logged. We show that a fully instrumented Palm can run up to two orders of magnitude slower. However, disabling a few popular calls and using a faster PDA adds an extra 50% overhead for interactive tasks. The mechanism also has limitations in collecting logs for system calls that are needed by the collection mechanism itself. Our logging mechanism works for about 80% (707 of 881) of Palm OS system calls. Palmist can be utilized by system practitioners to study the effects of their policy decisions on mobile users.

The Palmist tool is available online at http://greenhouse.cs.uga.edu/.

Acknowledgments

Carla Ellis suggested the need and requirements for Palmist. We also thank her for lending Palm devices during the initial phases of development. This work was supported in part by a research grant from the Yamacraw initiative.

References

[1] PalmOS SDK Version 3.5. http://www.palmos.com/dev/tech/tools/sdk35.cgi.

[2] Todd L. Cignetti, Kirill Komarov, and Carla Schlatter Ellis. Energy estimation tools for the palm. In *ACM MSWiM 2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.

[3] Jason Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 2–10, February 1999.

[4] Greg Hewgill et al. Palm OS Emulator (POSE). http://www.palmos.com/dev/tech/tools/emulator/.

[5] Edward Keyes. Hackmaster. http://www.daggerware.com/hackmstr.htm.

[6] Jacob Lorch and Alan J. Smith. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000. http://msdn.microsoft.com/msdnmag/issues/1000/VTrace/VTrace.asp.

[7] Robert Muth, Scott Watterson, and Saumya Debray. Code specialization using value profiles. In *Static Analysis Symposium*, pages 340–359, July 2000.

[8] Mark Newman and Jason Hong. A look at power consumption and performance on the 3com palm pilot. http://guir.cs.berkeley.edu/projects/p6/finalpaper.html, May 1998.

[9] NPD INTELECT PDA press update. Top two PDA brand shares. October 2000 - February 2001. http://www.intelectmt.com/corp/intelectmt/press/press-it/press˙010330.h%tm, March 2001.

[10] Palm Inc. Hotsync technology. http://www.palm.com/support/hotsync.html.

[11] Palm Inc. Palm device. http://www.palm.com/.

[12] Pine tree software. Pinemark benchmark. http://www.pinetreesoftware.net/palmos/pinemark/.

[13] N. Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA*, pages 95–106, 2000.