A Trace-Driven Simulator For Palm OS Devices

Hyrum Carroll, J. Kelly Flanagan, Satish Baniya
Computer Science Department, Brigham Young University
hyrum@pel.cs.byu.edu, kelly_flanagan@byu.edu, sbaniya@cs.byu.edu

Abstract

Due to the high cost of producing hardware prototypes, software simulators are typically used to determine the performance of proposed systems. To accurately represent a system with a simulator, the simulator inputs need to be representative of actual system usage. Trace-driven simulators that use logs of actual usage are generally preferred by researchers and developers to other types of simulators to determine expected performance.

In this paper we explain the design and results of a trace-driven simulator for Palm OS devices capable of starting in a specified state and replaying a log of inputs originally generated on a handheld. We collect the user inputs with an acceptable amount of overhead while a device is executing real applications in normal operating environments. We based our simulator on the deterministic state machine model. The model specifies that two equivalent systems that start in the same state and have the same inputs applied, follow the same execution paths. By replaying the collected inputs we are able to collect traces and performance statistics from the simulator that are representative of actual usage with minimal perturbation.

Our simulator can be used to evaluate various hardware modifications to Palm OS devices such as adding a cache. At the end of this paper we present an in-depth case study analyzing the expected memory performance from adding a cache to a Palm m515 device.

1 Introduction

Before a new computer system is implemented in hardware, developers often use software simulators to predict the performance of the new design. A detailed system simulator subjects a virtual system to representative workloads for accurate evaluation. Often a *trace*, or a sequence of system events recorded from an actual system, is used as input to the simulator. Simulators that use traces for input are known as *trace-driven simulators*.

Developers and researchers have demonstrated the utility of trace-driven simulation for desktop and server systems. They have used this technique to evaluate memory hierarchies, processor performance, etc. [16, 3, 17, 24, 19]. Rather than working on solutions to the well-known problems of collecting traces for desktops and servers [6, 10, 2, 8], we undertook to collect traces in an interactive and mobile environment.

Since Palm OS is used on more handheld devices than any other operating system [13], we focused on collecting traces for Palm OS devices. These devices are handheld computers running the Palm OS, which has a "preemptive multitasking kernel that provides basic task management" [1]. Tens of millions of handhelds around the world run Palm OS, with over a million units shipped in the first quarter of 2004 alone [13].

To collect these traces is a multi-step process. First, we instrumented a Palm OS handheld to record external inputs, referred as an *activity log*. Next, we transfer this log to a desktop computer. Using a simulator that we developed, we *replay* the collected activity logs. We use the terms replay and *playback* interchangeably to refer to the simulator's reproduction of an execution of events from an activity log. Furthermore, we instrumented the simulator to collect traces and performance metrics during playback.

One of the key issues with a monitoring system is the amount of perturbation introduced. An ideal monitoring procedure would not perturb the system at all. Quantifying the perturbation helps to determine how close to ideal a given monitoring system is. In this paper we quantify the amount of perturbation with the amount of overhead introduced, where overhead is the difference in execution time between the instrumented system and the original system. If the monitoring procedure alters the way in which a user operates a system, for example, by perceivably increasing the amount of time required to process inputs, then the overhead is unacceptable. On the other hand, if a system can be monitored in such a way that the user can not perceive the introduction of the monitoring system, then the amount of the overhead is acceptable.

1.1 Previous Work

Gannamaraju and Chandra's work with Palmist [9] provided a foundation for our work of collecting user's inputs. They collected system activity on mobile handhelds

by recording the occurrence of 80% (707 of the 880) of the Palm OS 3.5 system calls with *hacks*. (In the Palm OS community, hacks explicitly refer to system extensions, and are described in detail in section 2.3.2.) Due to their collection technique, Gannamaraju and Chandra were unable to generate a complete sequence of system activity. Since they recorded the occurrence of most of the system calls, their method introduced a large amount of overhead. They reported that the time required for each system call to execute increased by two or more orders of magnitude with Palmist running compared to a system not running Palmist. This amount of overhead is unacceptable. Furthermore, their technique requires significant memory requirements. For example, they generated 1.34 MB of records on the handheld to perform a set of tasks that requires about one minute of execution without Palmist running. Given that the maximum amount of memory found on Palm OS devices for which they did their work is between 8 and 16 MB, only a few minutes of execution could be traced. Using an eternal memory device to increase storage capacity would increase the already high amount of overhead.

Rose and Flanagan's work with CITCAT [21] inspired the model for our simulator. The CITCAT procedure defines the mechanisms used to collect and replay an initial memory state image and important events. Rose and Flanagan implemented CITCAT and produced *complete* instruction traces of desktop workloads. Flanagan defined complete instruction traces to be "those traces containing all CPU generated references including those produced by interrupts, system calls, exception handlers, other supervisor activities, and user processes" [6]. To do this, they collected the "state of the processor, caches, main memory, ... hardware interrupts, DMA, and other asynchronous or peripheral events that influence the processor"[21] with the aid of a hardware monitor such as a logic analyzer. They then initialized the simulator with the collected initial machine state image, and replayed the asynchronous events. During playback, they collected performance data and instruction and address traces. In their paper, Rose and Flanagan demonstrated that CITCAT can collect complete traces of desktop workloads. However, they used a large and expensive logic analyzer which is impractical for real-time mobile collection.

External hardware monitors, such as logic analyzers and oscilloscopes, significantly alter the user's behavior by restricting the mobility of the user. To use an external monitor requires connecting probes to various pins to gather data. Handhelds monitored with such equipment have to be handled very carefully to keep the probes attached. Furthermore, attaching a logic analyzer or oscilloscope to a handheld would greatly restrict the mobility of the device.

Several other studies besides Palmist have been performed on handheld devices, but have focused on evaluating the energy consumption and efficiency. Flinn and Satyanarayanan [7] collected information about handheld devices using oscilloscopes. Cycle-accurate simulators [23, 25, 14] have also been used to estimate energy consumption of handheld devices. For a simulator to produce accurate estimations, they need to be fed input representative of real usage. Cignetti, Komarov and Ellis' work [5] is unique in that they utilize a modified version of the Palm OS Emulator. They limited their studies to energy consumption estimates.

1.2 Our Solution

Our approach unites Rose and Flanagan's CITCAT [21] and Gannamaraju and Chandra's Palmist [9]. We borrow Rose and Flanagan's fundamental model of representing a computer system as a deterministic state machine and apply it to handheld devices. They showed that determining the execution path of a state machine, or a computer system, requires the initial state and sequence of inputs. We collect the initial state of a mobile devices by storing its memory contents on a desktop computer. To collect the sequence of inputs we use a collection technique similar to that used by Gannamaraju and Chandra, but with orders of magnitude less overhead. To illustrate the usefulness of our simulator, we present a case study demonstrating the expected performance of adding cache memory to a handheld device. This case study would not have been possible with previous data collection techniques for handheld devices.

1.3 Outline

The rest of this paper proceeds as follows: Section 2 explains the architecture of our system by first describing it from a high-level perspective, and then elaborating on the different components. The collection of the initial state, activity logs and their playback will all be covered in this section. Section 3 illustrates two different methods employed to validate the system. In the first method, we compare the activity log of the user's session and that of the emulated session. For the second method, we compare the final state of an emulated session with the final state of the handheld. In Section 4 we demonstrate the usefulness of the simulator and present the expected cache memory performance for a Palm m515. The final section concludes the paper and describes future work.

2 System Architecture

This section explains how we collect external inputs from a Palm OS device and use this data as input to our trace-driven simulator. We first explain our model, in general terms, then we describe our implementation of it. The model is divided into three components. First, the model requires the initial state of the system (comprised of the memory, processor, etc.). The second component is the sequence of inputs to the system. We refer to the log of user inputs,

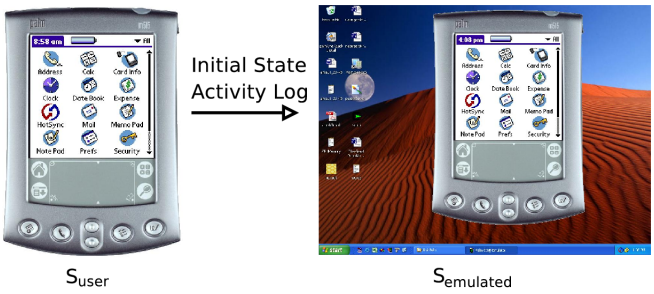

Figure 1. Application Of The Deterministic State Machine Model.

along with the corresponding timing information, as an *activity log*. The final component is an emulator which starts in the initial state and processes the inputs to the system. We then explain and evaluate the implementation of each of these components.

2.1 High-Level Overview

Since a computer processor is deterministic, we can use the deterministic state machine model to represent it, as did Rose and Flanagan [21]. First, let β be an initial state and δ be a sequence of external inputs. The model specifies that every time a deterministic state machine starts in β and δ is applied, the machine always follows the same sequence of states and ends in the same final state. Furthermore, if machine *A* and machine *B* are equivalent deterministic state machines, and both start in β, and δ is applied to both machines, then both machines *A* and *B* will follow the same path of execution and end in the same state.

The deterministic state machine model applies to situations in which the workload activity performed on system S_{user} is replicated on system $S_{emulated}$ (see Figure 1). First, let S_{user} be a system instrumented to record the initial state and all external inputs. Also, let $S_{emulated}$, be an equivalent system (e.g., an emulator or simulator) of S_{user} that can replay the events from S_{user} in an environment in which measurements can be made. At the end of a *session* (the period of time that inputs are collected) we transfer the initial state and collected inputs from S_{user} to $S_{emulated}$. Then $S_{emulated}$ is started in the initial state of S_{user} for that session. $S_{emulated}$ then processes and replays the collected activity log from S_{user}. Performance measurements and workload traces can be generated internally or externally on $S_{emulated}$. Since statistics and data are gathered on $S_{emulated}$ while the activity log is replayed, they represent the original execution by the user without system perturbation.

For our implementation of the model, S_{user} is a handheld computer operated by a human. We instrumented the handheld to collect the initial state and user inputs (i.e., pen movements and button presses). We then transferred the data for a session to a desktop computer. $S_{emulated}$ is a Palm OS device emulator that accurately models handheld computer devices. We execute the emulator on a desktop computer and use the initial state and activity log from S_{user} for playback. During playback, the emulator follows the same sequence of states as the handheld. By applying the deterministic state machine model to handhelds, we can collect the initial state and inputs of a device and replay them with an emulator on a desktop computer.

The following steps represent the chronological order of the methods we used to collect the relevant information from a handheld and replay it on an emulator:

- Instrument a handheld to collect user inputs
- Transfer the initial state of a handheld to the desktop
- Start collecting inputs
- Allow the user to operate the handheld normally
- Transfer the activity log from the handheld to the desktop
- Load the emulator with the collected initial state of the handheld
- Collect processor information while replaying the activity log on the desktop

We explain each of these steps in the following sections.

2.2 Initial State Collection

Before we give a handheld to a volunteer user, we instrument it to collect user inputs, capture its initial state, and transfer the state information to a desktop computer. We use ROMTransfer.prc, an application freely distributed by PalmSource Inc., to transfer a flash image from a Palm OS device to a desktop with a USB cable. To get the necessary contents of the RAM, we set the backup bit for all the applications and databases, and perform a HotSync.HotSyncing synchronizes the programs and databases on a handheld with a copy on the desktop with a USB cable.

The initial state of a device also includes the current state of the processor. Instead of recording all of the processor's registers we simply chose to start every session directly after a soft reset. This is acceptable because the processor follows a deterministic set of steps during a reset which can be replayed during playback.

2.3 Activity Logs

The deterministic state machine model specifies that in addition to the initial state, a complete sequence of external inputs, or *activity log*, is needed to accurately replay a session. For an activity log to be complete for a handheld, all forms of inputs must be considered. An activity log is simply a record of the time an external input occurred, the type of input and any relevant data necessary for playback.

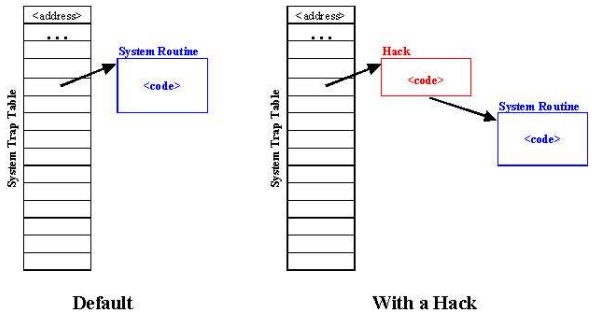

Figure 2. The Execution Path With And Without A Hack.

2.3.1 Handheld Inputs

Handheld computers are naturally interactive and allow for different forms of input from users. A typical handheld has the following forms of input: digitizer (touch screen), buttons, real time clock (RTC), memory card, IrDA/Serial Port and a reset button. Of these inputs, we collectively record stylus movements on the digitizer (touch screen), button presses (up, down, power, HotSync cradle and four application buttons) and the real time clock. These three input types account for the majority of input on handhelds. Also, the insertion, removal, and name of a memory card can be detected with our technique. We have chosen not to use memory cards in this study due to the extra complexity and requirements in storing activity logs and simulation. Allowing memory card to be used would require either storing the contents of the memory card that were accessed (and the timing of such events) or the entire contents or the memory card and simulating that interface. Memory cards that range into the gigabytes can be used with Palm OS devices. Due to time constraints and an expected increase in overhead, IrDA and serial port activity are not collected for similar reasons as the memory card. For simplicity, the reset event was not implemented in the simulator. Recording the event of a system turning off and on again has inherit problems that we have left for future work.

2.3.2 Hacks

In the Palm OS community, the word *hacks* has a specific meaning. A hack is a section of code contained in a routine that is "called in addition to or in lieu of the standard Palm OS routines" [12]. By default, at the time a system routine is called, the operating system looks up its address in the trap dispatch table, jumps to that location and continues execution. After the system routine finishes, execution continues in the application that called the routine. Inserting the address of a hack into the trap dispatch table forces the system to call the hack instead of the default system routine (see Figure 2).

Unlike Palmist, which uses a hack for every system call, our simulator only requires one hack for every system routine that processes user input. We wrote five hacks to patch the following system routines: EVTENQUEUEKEY, EVTENQUEUEPENPOINT, KEYCURRENTSTATE, SYSNOTIFYBROADCAST and SYSRANDOM. Each of these hacks opens a common database, inserts a record with the current tick counter and the real time clock values, the event type and any necessary data. It then closes the common database. Each hack also makes a call to the original system routine.

2.3.3 Evaluation of Activity Logs

The overhead introduced by collecting the activity logs, in terms of storage requirements and execution time, is negligible. The individual records each consume twelve or sixteen bytes on the handheld.

We found that volunteer users could operate a device normally for more than a week without having the database reach the maximum number of records (65,536). Gannamaraju and Chandra describe an implementation to overcome this barrier at the expense of additional execution time [9]. If the database contains the maximum number of the largest size records, it would require a total of 1536 KB of memory for the records and the database header information.

We quantitatively measured the overhead of the EvtEnqueuePenPoint hack by counting the number of pen events per second in the database with the stylus continuously pressed against the screen. The test was implemented on a Palm m515, which samples pen movements 50 times a second. The database initially contained no records. The device recorded an average of 50.0 pen events per second in the database indicating no perceptible overhead for pen sampling.

We also designed a test that called a hack in a tight loop on a handheld to further quantify the overhead. The test eliminated the call to the original system routine to isolate the overhead associated with the hack. Figure 3 illustrates the average execution time per call of the EvtEnqueueKey hack. This test revealed that the overhead of each call is proportional to the size of the database. For example, the average overhead for the EvtEnqueueKey hack when the database had zero to 10,000 records was 6.4 ms (the test with the stylus continuously pressed against the screen falls into this range). However, when the database had 50,000 to 60,000 records, the average overhead was 15.5 ms per call. Given that the position of the stylus is sampled every 20 ms, the overhead introduced by a hack when the database is nearly full is significant. The increase in overhead as the database size increases is believed to be caused by the OS's memory manager.

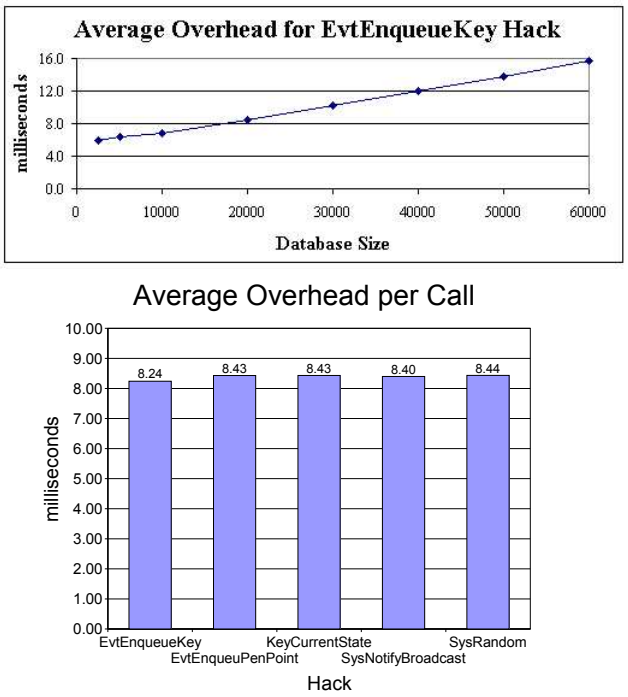

Figure 3. Average Overhead For The EvtEnqueueKey Hack And Each Hack Individually.

To keep the overhead at an acceptable level, we limited sessions to shorter time periods (e.g., two to three days). Figure 3 depicts the average overhead of each hack averaged over the first 30,000 iterations. Limiting sessions to two or three days generally kept the number of records well below 30,000. The overhead varied only slightly for each hack.

2.3.4 Summary of Activity Logs

We wrote five hacks to record activity logs containing user inputs on handheld devices. These hacks record all pen and button activity produced by a user. Furthermore, when sessions are reasonably limited, our hacks have an acceptable amount of overhead of less then 10 ms per call.

2.4 Playback of Activity Logs

The initial state and an activity log provide very little information in and of themselves. Their purpose is to provide the necessary information to replay the workload for the time period they were collected on a handheld. As stated in Section 2.1, an equivalent system can be used to start in the same initial state and replay the same inputs. This equivalent system can then be used to provide statistics and performance information without disturbing the integrity of the original workload. Our tool utilizes the Palm OS Emulator, POSE, as our equivalent system.

In this section, we first introduce POSE. We then explain the modifications we made to it. Next, we provide details of how we setup our simulator for playback. Finally, some limitations are described.

2.4.1 The Palm OS Emulator

The Palm OS Emulator, POSE, is a publicly available and widely used development tool for handhelds running Palm OS [18]. It runs on a desktop system, and emulates Palm OS based systems. POSE does this by keeping track of the processor state and fetching instructions out of a flash image and an allocated RAM. Stylus and button presses are simulated with the mouse. The emulator generates an event record to process the input. This event record is returned to the system when EVTGETEVENT is called (typically in the application's event loop). Our simulator is a modified version of POSE 3.5 that is instrumented to replay activity logs and collect performance data.

2.4.2 Modifications To POSE

We first modified POSE to generate stylus movements and button presses from an activity log. During initialization POSE reads a parsed activity log file and divides it into three groups — a list of events to be replayed synchronously with the tick counter and two queues. One queue is for KEYCURRENTSTATE key bit fields and the other is for random seeds (generated by non-zero SYSRANDOM calls).

To ensure the correct timing of the synchronous events, the emulated system's tick counter is checked to see if it is greater than or equal to the tick timestamp of the next event. If it is time for the next event, the emulator simulates the event and the emulator continues to execute the same instructions as the handheld. The emulator then continues execution until the next event is scheduled to occur. This process continues until the synchronous event log is exhausted.

The KEYCURRENTSTATE key bit field and random seed queues are utilized whenever their respective system calls are made. To assure proper execution, the emulator handles calls to the system routines KEYCURRENTSTATE and SYSRANDOM differently. For the routine KEYCURRENTSTATE, the emulator executes the function, then looks up the appropriate key bit field to return based on the emulated tick timer and the queue elements' tick timestamps. The emulator handles calls to SYSRANDOM in much the same way when its parameter is non-zero. The difference is the seed value from the queue is queried before SYSRANDOM is called. The parameter is overwritten with the seed value from the queue and execution continues. When the parameter for SYSRANDOM is zero, SYSRANDOM is ex-

ecuted as normal, returning the next random number from the existing seed's sequence.

We further modified POSE to track and output statistical execution information such as opcodes and memory references. To record the opcodes, we treated each executed opcode as an index into an array, and incremented the respective array element. Furthermore, we modified POSE *Profiling* functions to track the memory references. These values can be written to a file. We modified POSE to turn Profiling on during playback. When Profiling is enabled, POSE's native speed optimizations are ignored in favor of the original Palm OS code. A simple example of this involves the TRAPDISPATCHER. A Palm OS complier inserts Trap instructions for the system to handle system calls. When a Trap instruction is executed, and Profiling is enabled, the "TrapDispatcher looks at the program counter that was pushed onto the stack, uses it to fetch the dispatch number following the TRAP opcode, and uses the dispatch number to jump to the requested ROM function. [When Profiling is disabled,] the emulator essentially bypasses the whole TrapDispatcher function and . . . does the same operations with native x86, PPC, or 68K code" [20]. If Profiling were not enabled, the emulator will have skipped executing several instructions that a physical device would have, invalidating the collected data.

2.4.3 Simulator Setup

To use the simulator to playback a session, we first specify the filename of the activity log to use in a plain text file. Next, we run the emulator. After the emulator is finished warming up, we import all of the applications and databases corresponding with the initial state of the specified session. We then reset the emulator to get it into the same processor state as when the activity log started. Finally, we tell the emulator to start replaying events.

2.4.4 Limitations

Although our simulator works extremely well for workloads with state-based applications, choosing the Palm OS Emulator for playback does come with some limitations with respect to timing. POSE simulates the processor's tick counter, but not the RTC. We approximated the RTC by using the amount of time elapsed on the emulator's host machine (i.e., a desktop) since the last event. All events contain a RTC timestamp. Furthermore, due to approximating the RTC and other fine-grained timing differences (e.g., memory reads and interrupts) the combination of POSE and our methods are not appropriate for timing-sensitive applications.

The remaining limitations originate from the failure to fully implement functions found on handheld devices such as the serial port and memory card. These limitations are addressed in Section 5.1, Future Work.

By using the Palm OS Emulator to replay activity logs and collect data, we gather information about the workload on the handheld. POSE is used to replay user's activity because it is accurate and easily distributable.

3 System Validation

In the previous section, we discussed the deterministic state machine model. This model guarantees that equivalent deterministic state machines that start in the same state and have the same inputs applied, follow the same execution path and end in the same state. To validate the accuracy of our trace-driven simulator we employed a two-fold approach. First, we compared the inputs recorded on a Palm OS device while executing a test workload with the inputs recorded during playback on the emulator. Next, we correlated the handheld's final state and that of the final emulated state. In both cases we used three different test workloads for validation.

3.1 Test Setup

We performed both of the validation tests on a Palm m515 with a 33 MHz Motorola Dragonball MC68VZ328 processor and 16 MB of RAM. We utilized X-Master [15] to manage the hacks (see Section 2.3.2). We loaded three additional applications to prepare the handheld to collect activity logs. The first two applications we wrote to create a common database (the activity log) and to set all the backup bits of the databases. The third application, FileZ, a file manager, is produced by nosleep software [4] and is widely distributed as freeware.

The initial state for the first test workload mirrors the default factory settings for the system, except for the presence of the above mentioned applications and database. The initial state of the second test workload is the same as the final state for the first test workload. Also, the initial state of the last workload is the same as the final state of the second workload.

To begin the simulations, we loaded the simulator with the initial state by importing the applications and databases for the respective session and reset the simulator. After the simulator finished resetting, we told the emulator to start replaying events. After the simulator completed playback, we obtained the activity log and the final emulated state by HotSyncing the emulated system.

3.2 Test Workloads

We used three test workloads that we setup as defined by the previous section. The first two workloads followed a predefined script of actions. The final workload illustrated playing a game of Puzzle.

3.3 Activity Log Correlation

To validate the simulator, we first verified that the inputs collected from the physical device were replayed on the simulator. To do this, we compared the activity log generated while executing a test case and the activity log created during playback. To obtain the activity log on the simulator, we imported our hacks and X-Master along with the other applications and databases. Since the simulator accurately emulates the handheld system, the hacks were executed just as they did on the handheld.

The activity log from the handheld and that of the emulated session correlate very well. Each pen event recorded in the original activity log also appeared in the emulated activity log with the same coordinates. Furthermore, all of the button events were accurately replayed. However, the events in the emulated activity log sometimes occurred in short bursts. A burst of events would occur slightly behind schedule (< 20 ticks), and then return to being replayed at exactly the right tick count. The bursts are believed to be due to the different threads of the simulator running at different times, delaying the core processing thread momentarily. For this same reason, the KeyCurrentState events are at times slightly off in the emulated activity log. Although the activity logs from the initial session and the emulated session do not perfectly match, they contain virtually the same inputs, retaining the integrity of the log.

3.4 Final State Correlation

To further validate the simulator, we compared the final state of a test session on a handheld and the final state of the emulated session. The correlation of the two final states is not a holistic validation, but reflects the aggregate of the events. Since the final state of a session is largely determined by the state of memory, we analyzed the databases that were transferred to the desktop via a HotSync at the end of a session.

On a Palm OS device, applications are stored internally in the same format as record databases. Each database starts with a header, followed by the indexes of the records, and then the records themselves. The only structural difference of an application database and a record database is the presence of one or more code (or other) resources.

To quantify the differences between the final state of the handheld session and that of the emulated session, we compared the respective databases field by field. The databases correlated extremely well. The only exceptions are three fields entitled CREATIONDATE, LASTBACKUPDATE and MODIFICATIONDATE and the database named psysLaunchDB. The three fields represent the number of seconds since 12:00 A.M. on January 1, 1904 that the database was either created, backed-up, or modified, respectively. These three fields regularly differ between the two final states. The differences are attributed to the procedure of importing and exporting the databases to and from the simulator. The CREATIONDATE field for databases on the simulator was always zero, since the databases were imported instead of created. The LASTBACKUPDATE field was also always zero for the same databases because they were never backed-up on the emulated system. The MODIFICATIONDATE field for these databases was also zero unless it had a timestamp corresponding with the date of replay. Exporting the databases for the emulated session caused some databases to record the time of replay as the MODIFICATIONDATE. The differences in the timestamps in the database headers do not adversely effect the performance of the simulator.

The psysLaunchDB database is used solely by the operating system and has an unpublished format. We estimate from its name and raw contents that it stores information about applications that can be run from the *home* screen. The few single byte differences between the records of the two databases are also attributed to the procedure of loading databases into the simulator. We estimate that the execution of the system is not adversely affected from the differences in psysLaunchDB.

From our validation, the simulator replays activity logs very accurately. The very few differences are determined to not affect the integrity of the simulation, thus making it suitable for performance and analysis studies.

4 Cache Case Study

This section presents an in-depth case study analyzing the effects on memory performance from adding different caches to a Palm OS device. In this section we present what we believe to be the first cache simulation results for devices running Palm OS.

4.1 Introduction

Over the past decades, the gap between processor and memory performance has widened. Hennessy and Patterson report that processor performance has increased annually by 55% since 1986 while memory lags behind with an increase of 7% per year [11]. To reduce this gap, memory hierarchies have been designed to take advantage of locality to maximize performance and minimize cost.

A simple memory hierarchy includes a CPU, a *cache* and memory. A cache is the memory closest to the CPU and contains a subset of the information found in memory, but is designed to respond more quickly.

When the processor requests the contents of a memory location from a cache, and that information is found, it is called a *hit*. Conversely, if the contents are not found in the cache, it is called a *miss*. When a miss occurs, a *block* is read from memory, placed in the cache, and returned to the processor. A block is typically in the tens of bytes, and

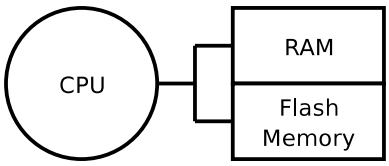

Figure 4. The Memory Hierarchy For The Device We Investigated, A Palm m515.

Session	RAM Refs (Ms)	Flash Refs (Ms)	Events	Elapsed Time	Ave Mem Cyc
1	214	443	1243	24:34:31	2.35
2	31	69	933	48:28:56	2.38
3	34	76	755	24:52:55	2.39
4	234	486	1622	141:27:26	2.35

Table 1. Volunteer User Session Data.

contains the contents of the desired location and adjacent addresses.

When a cache miss occurs, an existing block must be displaced to make room for the requested block. The most common algorithm for selecting the block to replace is the *least-recently used* (LRU) algorithm.

Average effective memory access time is used to quantify the performance of various cache configurations and is described by the following equation:

$$T_{eff} = T_{hit} + MR \cdot T_{miss} \tag{1}$$

T_{hit} is the time (in CPU clock cycles) that a cache requires to service a cache hit. *MR*, the *miss rate* of the cache, specifies the percentage of requests that result in a miss. T_{miss} is the time (in CPU clock cycles) that a memory hierarchy requires to deliver the desired block on a cache miss.

Since Palm OS devices have both RAM and flash memory (see Figure 4) the average effective memory access time for a Palm OS device is computed with the following equation:

$$T_{eff} = T_{hit} + MR\left(\frac{REF_{RAM}}{REF_{total}}T_{RAM\_miss} + \frac{REF_{flash}}{REF_{total}}T_{flash\_miss}\right) \tag{2}$$

where $REF_{total} = REF_{RAM} + REF_{flash}$. T_{RAM_miss} and T_{flash_miss} are the CPU cycles required to fetch a block from the RAM and flash respectively. REF_{RAM} and REF_{flash} are the number of RAM and flash references.

The following equation is used to calculate the average effective memory access time for a system without a cache:

$$T_{eff} = \frac{REF_{RAM}}{REF_{total}}T_{RAM\_miss} + \frac{REF_{flash}}{REF_{total}}T_{flash\_miss} \tag{3}$$

With a handheld device, not only is performance important, but power consumption is as well. The system performance of a device is the time required to perform a task. With a handheld, power consumption defines the amount of time a device can be used. Studies have also been done to show that adding a cache not only increases performance but can reduce the battery consumption for portable devices [22].

4.2 Experimental Setup

Table 1 summarizes the four sessions used in our cache simulations. *Events* refers to the number of entries in each of the activity logs. *Elapsed Time* is the amount of time (in HH:MM:SS) each session spans. *Ave Mem Cyc* is the average effective memory access time (in cycles), without a cache as computed with Equation 3.

The volunteer user operated a Palm m515 device with 16 MB of RAM, 4 MB of flash and a 33 MHz Dragonball MC68VZ328 processor (which does not have a cache). Before each session, we initialized the device, and collected the necessary information for the simulator, as described in Section 2. While replaying a session with the emulator, we collected a log of the memory requests. We used these logs, or *traces*, with a cache simulator that we wrote to obtain cache miss rates. We simulated 56 different cache configurations by varying the cache size, line size and associativity. The LRU replacement policy was used in every configuration.

4.3 Cache Simulation Results

Figure 5 illustrates miss rates for the 56 cache configurations of one session. These results are typical of the other sessions in Table 1. Caches with a line size of 32 bytes performed better than those with 16 byte lines except for the largest cache sizes simulated with 4 and 8 way set associativities. Furthermore, increasing the associativity typically decreases the miss rate.

The Dragonball MC68VZ328 requires one cycle for RAM accesses and three cycles for flash accesses.With the flash contributing about two thirds of the total references, the average effective memory access time is closer to that of a flash access time than a RAM access time (see Table 1). We used Equation 2 with T_{hit} equal to one cycle and T_{RAM_miss} and T_{flash_miss} equal to one and three cycles respectively to compute the average effective memory access times. Figure 6 displays the results for the same 56

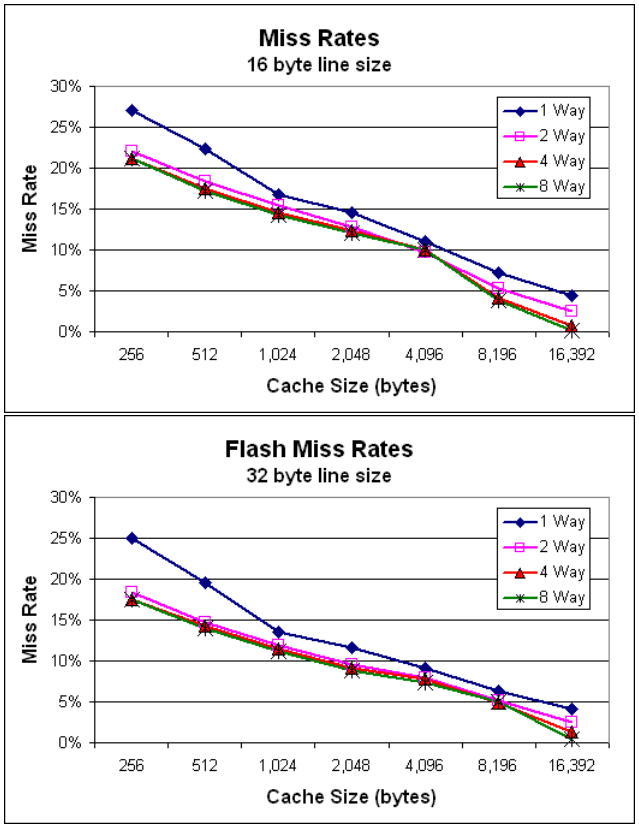

Figure 5. Miss Rates For 56 Cache Configuration.

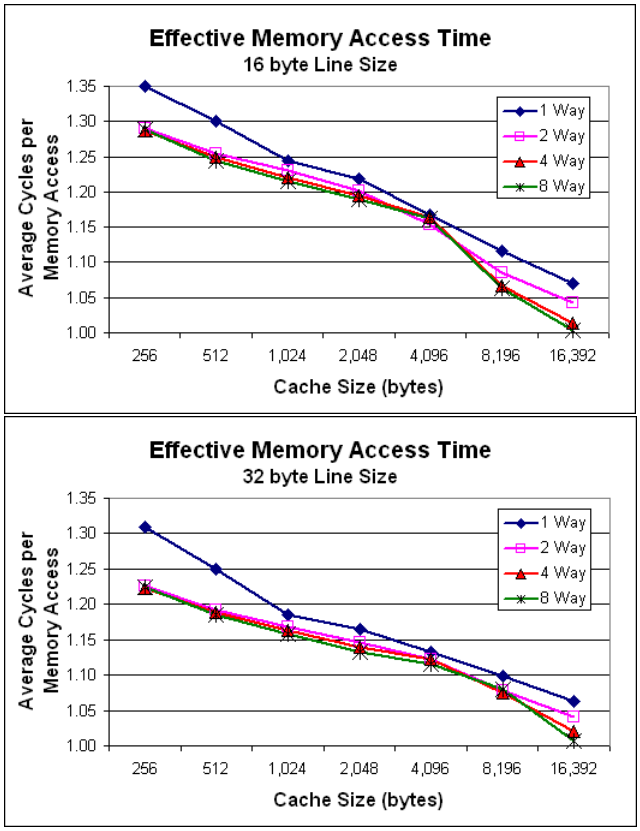

Figure 6. Average Effective Memory Access Times.

cache configurations. In all configurations, adding a cache significantly reduces the average memory access time.

The small cache sizes used in this study exhibit the same miss rate trends found in larger caches used in desktop systems. Figure 7 shows the miss rates for a desktop system, exemplifying these trends. This sample comes from the Trace Distribution Center at Brigham Young University, a repository of desktop and server traces.

4.4 Conclusion

Average memory access times for current Palm OS workloads suffer from requests to the flash dominating those to the RAM, which require three times as many cycles to complete. Our cache simulations show that even relatively small caches can reduce the effective memory access time by 50% or more! This is mostly due to the flash memory receiving the majority of references. Adding a cache to a Palm OS device using a Dragonball MC68VZ328 processor can greatly reduce the average effective memory access time and potentially reduce the battery consumption.

5 Conclusion

Hardware prototypes are expensive to build. To minimize the overall development cost of new hardware, system architects rely on simulation to guide development choices. Trace-driven simulators provide more accurate performance estimates than other simulation approaches.

We have designed a trace-driven simulator for Palm OS devices. First, the initial state is collected from a handheld device, then the device is instrumented to record external inputs with five different hacks. The log of external inputs, or activity log, is collected while an actual user is running real applications under normal operating conditions. After transferring the activity log to a desktop system, we replayed it on an equivalent system, instrumented to record opcode usage, memory references and performance statistics. With this data, tests such as energy consumption and cache simulations can be realistically and accurately performed. Furthermore, our cache simulations show that adding even a small cache memory to a Palm OS device can greatly reduce the average effective memory access time.

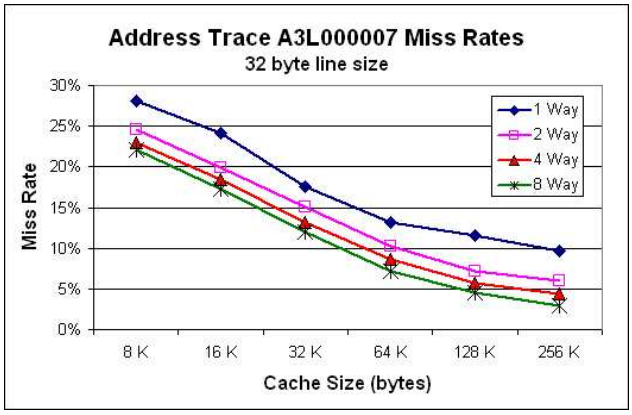

Figure 7. Miss Rates For A Desktop Address Trace.

5.1 Future Work

While our simulator provides statistical performance metrics for the most commonly used features of Palm OS devices, it does not currently replay activity logs that involve memory cards, IrDA or serial port activity, resets or battery information. Each of these features have been left for future work and would allow a more diverse group of sessions to be replayed with the simulator. Furthermore, a more comprehensive study of usage behavior with more users over a longer time frame would provide more accurate estimates for cache simulations and other such tests.

References

[1] C. Bey, E. Freeman, G. Hillerson, J. Ostrem, R. Rodriguez, and G. Wilson. *Palm OS Programmer's Companion Volume I*. PalmSource, 1996-2001.

[2] A. Borg, R. E. Kessler, G. Lazana, and D. W. Wall. Long address traces from risc machines: generation and analysis. Research Report 89/14, Sept 1989.

[3] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Proc. of 17th Int. Symp. on Computer Architecture*, pages 270–279. ACM, 1990.

[4] T. Bulatewicz. The nosleep software open source project. Available: http://nosleepsoftware.sourceforge.net/ [accessed April 2004], September 2002.

[5] T. L. Cignetti, K. Komarov, and C. S. Ellis. Energy estimation tools for the palm. In *ACM MSWiM 2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Aug. 2000.

[6] J. K. Flanagan, B. E. Nelson, J. K. Archibald, and K. Grimsrud. Incomplete trace data and trace driven simulation. In *Proc. of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS*, pages 203–209. SCS, 1993.

[7] J. 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, Feb 1999.

[8] R. M. Fujimoto and W. C. Hare:. On the accuracy of multiprocessor tracing techniques. Report GIT−CC−92−53, Georgia Institute of Technology, June 1993.

[9] R. Gannamaraju and S. Chandra. Palmist: A tool to log palm system activity. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization WWC4*, pages 111–119, Dec. 2001.

[10] K. Grimsrud, J. K. Archibald, B. E. Nelson, and J. K. Flanagan. Estimation of simulation error due to trace inaccuracies. IEEE Asilomar Conference, 1992.

[11] J. L. Hennessy and D. A. Patterson. *Compuuture Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3 edition, 2003.

[12] E. Keyes. The hackmaster protocal. Available: http://www.daggerware.com/hackapi.htm [accessed 2001].

[13] T. Kort, R. Cozza, L. Tay, and K. Maita. Palmsource and microsoft tie for 1q04 pda market lead. Technical report, Gartner Dataquest, April 2004.

[14] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded RISC processors. In *LCTES/OM*, pages 1–10, 2001.

[15] LinkeSOFT. X-master 1.5 free extension (hack) manager. Available: http://linkesoft.com/xmaster/ [accessed 2002].

[16] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *In Proc. of 4th Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 75–84. ACM, 1991.

[17] O. A. Olukotun, T. N. Mudge, and R. B. Brown. Implementing a cache for a high-performance gaas microprocessor. In *Proc. of 18th Int. Symp. on Computer Architecture*, pages 138–147. ACM, 1991.

[18] PalmSource Inc. Palm os, palm powered handhelds, and 18,000 software applications. Available: http://www.palmsource.com [accessed July 2001].

[19] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance−optimal multi-level cache hierarchies. In *Proc. of 16th Annual Int. Symp. on Computer Architecture*. IEEE, 1989.

[20] K. Rollin. The Palm OS Emulator. *Handheld Systems*, 6(3), May/June 1998.

[21] C. Rose. Citcat: Constructing instruction traces from cache-filtered address traces. Master's thesis, Brigham Young University, April 1999.

[22] J. Su. Cache optimization for energy efficient memory hierarchies. Master's thesis, Brigham Young University, 1995.

[23] N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *ISCA*, pages 95–106, 2000.

[24] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proc. of 1991 ACM Sigmetrics*, pages 79–89. ACM, 1991.

[25] W. Ye, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Design Automation Conference*, pages 340–345, 2000.