A Checkpointing Tool for Palm Operating System

Chi-Yi Lin and Sy-Yen Kuo
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
sykuo@cc.ee.ntu.edu.tw

Yennun Huang*
PreCache Inc
555 Madison Avenue,
New York, NY 10022
yhuang@precache.com

Abstract

It is foreseeable that handheld devices will be involved in the arena of distributed computing in the near future. To provide a dependable computing environment, checkpointing and rollback recovery is a useful and important technique for fault-tolerant distributed computing systems. For the most popular platform among handhelds – Palm OS, its built-in HotSync tool can take a partial snapshot of a system state, but it synchronizes only the static data in the handheld with a PC. All dynamic data of applications are lost if a failure occurs and the Palm OS is reset. In order to accommodate mobile computing devices with checkpointing and rollback recovery capability, dynamic data such as global variables should be checkpointed to tolerate system reset/crash failure. Therefore, we developed a checkpointing tool, which provides a set of APIs to checkpoint Palm applications. Using the checkpointing tool, dynamic data in a Palm device can be saved and recovered from a system reset. In this paper, we describe the tool and demonstrate its usefulness in four popular Palm applications.

1. Introduction

As computer and wireless communication technology advance, not only the computing capability of palm-sized computers is getting more powerful, but also more and more handheld devices are becoming network-accessible. The observation leads to a definite trend: in the near future, handheld/mobile devices will be qualified to be participants in a distributed computation [1].

To make distributed computing viable, it is necessary to ensure the reliability of the computing environment. It is well known that *checkpointing and rollback recovery* is a useful and important technique for fault-tolerant distributed computing systems. In order to provide a robust computing environment composed of handheld/mobile devices, it is reasonable to apply checkpointing and rollback recovery mechanism to these gadgets. Among the popular handheld devices, a recent study shows that up to 75 percents of them are equipped with the *Palm OS* [2]. Therefore, we chose Palm OS as our targeting platform to provide a generic checkpointing tool for various applications.

The Palm OS divides the total available RAM space into two logical areas: *dynamic* RAM and *storage* RAM. The two memory areas are analogous to the RAM and the disk storage of a typical desktop system, respectively [3]. The data in the storage RAM can be synchronized with a PC via the built-in *HotSync* tool. That is, HotSync tool can take a partial snapshot of a system state. But all dynamic data of applications are lost if a failure occurs and the Palm OS is reset. From this point of view, Palm OS is not reliable enough to be used in a dependable distributed computation. If the data in dynamic RAM such as global variables can be checkpointed during program execution, the failures of system reset/crash can be tolerated. Therefore, we developed a checkpointing tool, which provides a set of APIs to checkpoint Palm applications. Using the checkpointing tool, dynamic data in a Palm OS device can be saved and recovered from a system reset. To the best of our knowledge, this is the first checkpoint and recovery tool in Palm OS.

The rest of this paper is organized as follows. Section 2 introduces the Palm OS and the concept of mobile computing with checkpointing. Section 3 describes the checkpointing mechanisms of our checkpointing tool. Section 4 demonstrates the experiments and gives some discussions. Finally, Section 5 concludes our work.

2. Preliminaries

2.1. Palm Operating System Overview

Palm OS runs on top of a preemptive multitasking kernel [4]. One task runs the user interface, while other tasks handle things like monitoring input from the hard buttons or handwriting area. Since the user interface permits only one application to be open at a time, the currently opened application has control over the entire screen. As a result, applications run within the single-user interface thread and therefore they cannot be multithreaded.

* The work was completed when the author was in AT&T Labs, NJ.

0-7695-1101-5/01 $10.00 © 2001 IEEE

Similar to other platforms, Palm OS applications are event driven. The application fetches events from the event queue and dispatches them appropriately [5].

2.1.1. Memory Architecture. Each Palm OS device has a memory module known as a *card* which contains ROM, RAM, or both [3]. The main suite of applications provided with each Palm OS device is built into ROM, while additional applications and system extensions can be loaded into RAM. The RAM store is divided into two logical areas: *dynamic* RAM and *storage* RAM. Dynamic RAM is mainly for temporary allocations, and storage RAM stores application programs and database records.

The entire dynamic RAM is used to implement a single heap – *dynamic heap*, which provides memory for dynamic allocations, including global variables, system dynamic allocations, application stacks, temporary memory allocations, and application dynamic allocations. From Palm OS 3.0 and above, the storage RAM is also configured as a single heap called *storage heap*.

In the Palm OS environment, all data are stored in memory *chunks*. A chunk is an area of contiguous memory allocated by the Palm OS *memory manager*. There are two kinds of chunks – *relocatable* chunks (called *handles*) and *nonrelocatable* chunks (called *pointers*). A relocatable chunk is a chunk that the memory manager can move it around as necessary in order to keep free memory space contiguous. Since the physical address of a handle may not be fixed, a *master pointer table* is responsible for translating a handle value to its physical address containing the chunk. For nonrelocatable chunks, there is no need to do address translation and thus the master pointer table contains no entry for them.

The *data manager* is responsible for managing memory in the storage heap [6]. The allocated memory that holds storage data is a *record* in a *database*. In Palm OS, a database is a list of memory chunks and associated database header information. Since that the storage heap is hardware write-protected, changes to the databases can only be made through the data manager APIs.

2.1.2. Event Loop. The Palm OS event loop gets events in the event queue and then handles events according to the event type. The loop terminates when it gets an appStopEvent from the event queue. Listing 1 shows a sample of a typical event loop.

The nested *if* statements constitute the main part of the loop. The first routine SysHandleEvent provides functionality common to all Palm applications such as the event that a hardware button is pressed. The second routine MenuHandleEvent handles events involving menus. The third routine ApplicationHandleEvent is responsible for loading forms and associating an event handler with the form. The last routine FrmDispatchEvent in the event loop indirectly provides form-specific handling, such as copy/paste in the text fields of a form, by calling the form's event handler.

```
static void EventLoop (void)
{
    EventType event;
    do {
        EvtGetEvent (&event, evtWaitForever);
        if (! SysHandleEvent (&event))
            if (! MenuHandleEvent (NULL, &event, &error))
                if (! ApplicationHandleEvent (&event))
                    FrmDispatchEvent (&event);
    } while (event.eType != appStopEvent);
}
```

Listing 1. Sample code of an event loop

2.1.3. System Resets. There are two different levels of system resets that impose different effects on the memory storage. A *soft reset* clears the entire dynamic heap, while leaving storage heap untouched [7]. The operating system restarts from scratch with new stacks, new global variables, restarted drivers, and a reset communication port. On the other hand, a *hard reset* not only clears the dynamic heap, but also erases the whole storage heap.

2.1.4. The Built-in HotSync Tool. Palm OS provides a built-in synchronization tool – *HotSync*, which synchronizes the storage RAM with a PC. Databases in the storage heap such as application programs and records are backed up in the PC, but *HotSync does not save and recover the dynamic heap*. For example, if the Palm OS device undergoes a *hard reset* that clears all contents in the RAM store, the applications installed can be restored from a HotSync synchronization, but the dynamic data in the dynamic heap cannot be recovered. To sum up, with only the HotSync tool, the Palm OS device will lose its current program state if a failure occurs accidentally.

2.2. Mobile Computing and Checkpointing

Mobile computing makes it possible for us to access and exchange information while we travel. Since mobile computing devices are getting more powerful, they will be able to participate in various applications of distributed computing, such as information gathering and processing. But mobile devices suffer from failures very easily, compared to fixed hosts connected with each other through hard-wired networks [8]. The failures include physical damage, power shortage, loss of network connectivity, or even devices got lost or stolen. As a result, a proper fault-tolerant mechanism is essential to ensure the reliability in a mobile computing environment.

It is well known that *checkpointing and rollback recovery* techniques provide fault-tolerant capability for distributed computing systems. A complete introduction to these techniques can be found at [9]. Recently, many checkpointing and rollback recovery protocols for mobile computing systems have been proposed [10-12]. These

protocols focus mostly on avoiding the need for a system-wide synchronization during checkpoint collection, while keeping the recovery procedure efficient. The storage management for checkpoints is also discussed [12] which makes best use of the limited storage space on base stations.

The essence of the dependable mobile computing using Palm devices is the checkpoint mechanism for Palm OS, which is described in the next section.

3. A Checkpointing Tool for Palm OS

3.1. The Basic Concept of Our Approach

In order to survive the dynamic data of a running application, it is necessary to save the data into a database in the storage heap, which is not cleared during system soft reset. The saved data is a record in a database, which serves as a checkpoint for later recovery. Also, through HotSync that synchronizes the handheld device with a PC, the checkpoint record can be further copied to the non-volatile storage on a PC. Once the handheld device undergoes system hard reset, it can recover the whole storage heap from synchronization with a PC, and then the specific application state can be recovered from the saved checkpoint.

3.2. Getting Bounds of Global Variables

Since most applications utilize global variables to save their application state, we have to capture and then restore these global variables in order to recover the application state. Global variables are allocated by Palm OS, and there is no built-in API to get the bounds of these variables. Our experiment shows that global variables are allocated sequentially in the dynamic heap, so we added two dummy variables that are declared as the first and the last global variable, respectively. By passing the bounds as parameters to our checkpointing API, we are able to save and then recover the correct range within the dynamic heap. Listing 2 shows part of the *MineHunt* application in which two dummy variables are declared.

```
// Private global variables
static UInt32 GlobalStart; // dummy variable 1 as the first
                              global variable
static GameType        Game;
static MinePrefType    MinePref;
static Int16 PieceBitmapTable[lastSquareGraphic] =
{
    CoveredSquareBitmap,
    ......
};
static UInt16              SoundAmp;
static UInt32 GlobalEnd;  // dummy variable 2 as the last
                              global variable
```

Listing 2. Two dummy variables declared to locate the bounds of global variables

3.3. Launching Checkpointing Tool

In Figure 1 (a), a menu item CkpTool/Launch is added in the *MineHunt* application that launches our checkpointing tool. The checkpointing tool is running as a subroutine of its caller; i.e., the execution of the caller application is suspended, but not quitted. This is done by calling a system function SysAppLaunch that launches a specified application as a subroutine of the caller [13]. SysAppLaunch takes four *in* parameters and one *out* parameter, among which the fourth *in* parameter is a memory pointer that points to the launch code parameter block. The launch code parameter block is utilized to pass the global variable bounds of the current application to our checkpointing tool. Figure 1 (b) shows the main form of the checkpointing tool. The first button takes a snapshot of the application's dynamic data; the second button performs recovery from the snapshot; the third button deletes the snapshot saved in the checkpoint database; and the last button terminates the checkpointing tool.

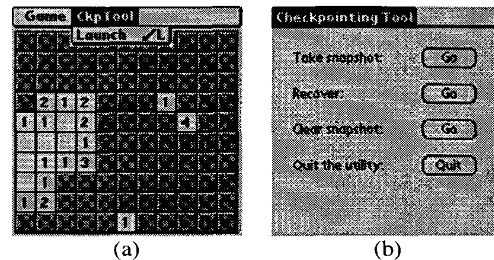

Figure 1. (a) A menu option that launches the checkpointing tool
(b) Main form of the checkpointing tool

Another system function SysUIAppSwitch can also be used to launch another application programmatically [13]. But using SysUIAppSwitch will terminate the currently running application because SysUIAppSwitch sends the current application an appStopEvent and then starts the specified application. This is not our case, since we need to resume execution of the checkpointed application after doing memory manipulations in our checkpointing tool.

3.4. Checkpointing Procedure

As Section 3.2 mentioned, we capture the global variable bounds of the current application by acquiring the addresses of the two dummy variables. By calculating the difference between the two addresses, we get the record size (globalSize in the source code) needed to save these global variables. Following is the procedure of taking a checkpoint. Firstly, the checkpointing tool looks up the storage heap if the checkpoint database exists. If the database is not found, the procedure creates a new database

and then proceeds. Secondly, the number of records in the database is checked; if none exists, a record of size globalSize is allocated. Finally, with the starting address of the global variable bounds (ParentGlobalBound.start), we use the system function DmWrite to do memory copy from dynamic heap to storage heap. Listing 3 shows the major part of the CkpTake procedure in our program.

```
// Compute the size needed to save the checkpoint.
globalSize =ParentGlobalBound.end-ParentGlobalBound.start+1;
// Get the checkpoint database; if not found, create one.
gDB = DmOpenDatabaseByTypeCreator (myDBType,
        myCreaterName, dmModeReadWrite);
if (!gDB) {
   error = DmCreateDatabase (0, myDBName,
           myCreaterName, myDBType, false);
   ErrFatalDisplayIf (error, "Couldn't create new database.");
   gDB = DmOpenDatabaseByTypeCreator (myDBType,
           myCreaterName, dmModeReadWrite);
}
// Get # of records in the database; if none, allocate one.
numRecordsInDB = DmNumRecords (gDB);
if (numRecordsInDB == 0) {
   UInt16 chunkRecNum = dmMaxRecordIndex;
   handle = DmNewRecord (gDB, &chunkRecNum, globalSize);
}
// Acquire a mem ptr to the record and do memory copy
recordH = DmGetRecord (gDB, 0);
recordP = MemHandleLock (recordH);
status = DmWrite (recordP, 0, (UInt32*)
          ParentGlobalBound.start, globalSize);
```

Listing 3. Sample code of the checkpointing procedure CkpTake

3.5. Recovery Procedure

As we can see, the bounds variable (ParentGlobalBound) of the checkpointed application is not saved along with the memory content in the checkpointing procedure. In fact, we don't need to save it because the bounds may shift at each time the application is loaded. The last given bounds might be invalid at this time when the application is running again, no matter if the device got reset or not. The correct procedure is to get the current global variable bounds of the application, pass it to the checkpointing tool, and then restore the checkpoint to the new bounds in the dynamic heap. It will lead to system crash if we restore the checkpoint to an invalid memory address. Figure 2 is an illustration of the scenario. When the checkpoint is taken, the global variable bounds couple is (x, y); when the state is being recovered, the bounds couple becomes (x', y').

The recovery procedure works as follows. Firstly, it acquires a reference to the checkpoint database. If the database cannot be found, a "database missing" alert is shown and the procedure quits. Secondly, it checks the number of records in the database. If the number is zero, it means no previous checkpoint exists, so a "no record found" alert is shown, and then the procedure quits. Finally, the system function MemMove performs memory copy from the checkpoint database to the current global variable bounds. Listing 4 shows the sample code.

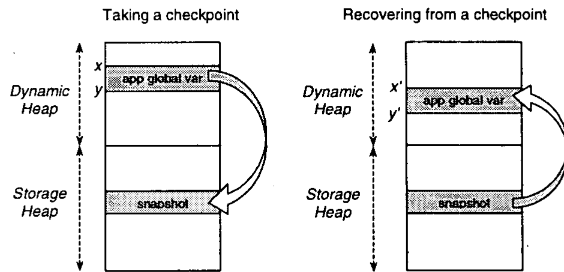

Figure 2. Illustration of bounds shift between checkpoint taking and recovering

```
// Open the ckpt db. If db is not found, display alert and return 0.
gDB = DmOpenDatabaseByTypeCreator (myDBType,
        myCreaterName, dmModeReadWrite);
if (!gDB) {
   FrmAlert(DBMissingAlert); return 0;
}
// Get # of records in the db. If # = 0, display alert and return 0.
numRecordsInDB = DmNumRecords (gDB);
if (numRecordsInDB == 0) {
   FrmAlert(NoRecordAlert); return 0;
}
// Get a ptr to the ckpt record, then overwrite the global bounds
recordH = DmGetRecord (gDB, 0);
recordP = MemHandleLock (recordH);
status = MemMove ((UInt32*) ParentGlobalBound.start,
          recordP, globalSize);
```

Listing 4. Sample code of the recovery procedure CkpRecover

3.6. Returning to the Caller Application

When the checkpointing tool as a subroutine quits, program execution returns to the caller application. At this point, we need to get the display of the caller application redrawn, because once the subroutine quits, the system loses its active form; i.e., the system has no idea which form it should load. So we send a "form load event" (frmLoadEvent) with the appropriate form ID to the event queue. Then we use another system function FrmUpdateForm to make an explicit redraw of the form that is loaded into memory. Listing 5 is the sample code that does form redraw.

```
if(dbID) //Call the checkpointing tool as a subroutine
   error = SysAppLaunch(cardNum, dbID, 0, ...);
// Assign frmLoadEvent type to myFormEvent, specify the form
  ID, then add it to event queue
myFormEvent.eType = frmLoadEvent;
myFormEvent.data.frmLoad.formID = MainForm;
EvtAddEventToQueue (&myFormEvent);
// Request to redraw the whole form
FrmUpdateForm (MainForm, frmRedrawUpdateCode);
```

Listing 5. Sample code handling form redraw after returning to caller application

4. Experiments and Discussions

4.1. Experiment Environment

For the hardware equipments, we use one Palm Vx and one Palm VIIx, both with 8MB RAM. For the software development environment, we use Metrowerks CodeWarrior for Palm OS R6 with SDK 3.5 support and Palm Emulator 3.0a7, both running on Microsoft Windows 98. The operating system on the Palm device is Palm OS 3.5. For all Palm OS function calls and important data structures, a complete reference can be found at [14].

4.2. Example Applications

Palm Inc. provides some application source codes that are downloadable through their website [15]. Among them we chose four popular games – *MineHunt*, *Puzzle*, *HardBall*, and *Raptoids* as our examples. The example source codes come with the format of CodeWarrior project files. In the source files, we added two dummy global variables and a menu option that launches our checkpointing tool. The database name of the tool is CkpLib and the compiled size is 5KB.

- **MineHunt** is similar to the Minesweeper in Microsoft Windows. The application state includes the status of each block, the number of mines left, game sound amplitude, and the preference of game difficulty. The checkpoint size is 497 bytes.
- **Puzzle** is a board game that the blocks with numbers need to be sorted in ascending order. The application state is the distribution of the 15 numbered blocks and the empty position. The checkpoint size is 23 bytes.
- **HardBall** is an animated game that a bouncing ball breaks the bricks on the top. The game state includes the position of the ball and the pad, the direction and speed of the ball, status of the bricks, game level, and the score. The checkpoint size is 459 bytes.
- **Raptoids** is a shooting game that the aircraft moves around and shoots the flying rocks. The game state includes the direction/position/speed of the aircraft and the rocks, the number of aircrafts left, the size of a rock, the positions of bullets, and the score. The checkpoint size is 2.897 Kbytes.

With very minor modifications to the source code of these example applications, our experiment shows that using the checkpointing tool, the program states of all the applications can be saved, and then be recovered correctly.

4.3. Evaluations

The overhead introduced by our checkpointing tool includes the time needed to take or recover from a checkpoint on a Palm OS device, and the power consumption of the checkpointing procedures.

The Palm OS device maintains a *system tick* count that starts at 0 when the device is reset [7]. The system tick is a *0.01-second timer*, which can be used to calculate the time required taking or recovering from a checkpoint. We get the result by comparing the tick counts returned by the system function TimGetTicks before and after the checkpointing procedure.

The initial result shows that it takes only one system tick, or 10 milliseconds, checkpointing or recovering for the four examples. To make the measurement more accurate, we modified the two procedures CkpTake and CkpRecover to be looped for 1000 times, and then we derived the average value. The result is shown in Table 1.

Table 1. The execution time of CkpTake and CkpRecover for the example applications

Example Applications	Checkpoint Size (Kbyte)	Time for CkpTake (millisecond)	Time for CkpRecover (millisecond)
MineHunt	0.497	9.25	8.70
Puzzle	0.023	8.92	8.19
HardBall	0.459	9.24	8.73
Raptoids	2.897	12.20	11.51

From Table 1, we can find that our checkpointing tool brings barely overhead to a relatively slow Palm OS device, when the checkpoint size is only a couple of Kbytes. For the case that the checkpoint size is much larger, we modified the checkpointing procedure to do memory copy for 1000 times continuously (only DmWrite and MemMove are looped), to simulate the condition. Table 2 lists the result when the *effective* checkpoint size is 1000 times larger for the four examples. It is shown that when the checkpoint size is as large as 3 Mbytes, it takes less than 4 seconds to finish checkpointing. From the perspective of time efficiency, we believe that our checkpointing tool is viable.

Table 2. The execution time of CkpTake and CkpRecover with large checkpoint size

Example Applications	Effective Checkpoint Size (Kbyte)	Time for CkpTake (second)	Time for CkpRecover (second)
MineHunt	497	0.98	0.59
Puzzle	23	0.45	0.08
HardBall	459	0.94	0.55
Raptoids	2897	3.66	3.15

The other important consideration is the power consumption of taking a checkpoint. According to [7], the

current battery level information can only be obtained through the SysBatteryInfo routine. So we use the system function to detect battery levels before and after taking a checkpoint. The returned value of the battery level is an integer, e.g., battery level 100 represents a fully charged battery, while battery level 35 represents that only 35% of the power is available. Since the battery level is not precise enough, our experiment shows that the battery level is not changed after taking a checkpoint. Even if we modified the code to make CkpTake be executed 10000 times continuously, the battery level is still unchanged. This means that taking a checkpoint consumes only very tiny amount of the power that we can hardly tell.

4.4. Discussions

Our tool provides a generic checkpointing mechanism for various Palm applications without checkpointing capability. Since it is required to modify the source code of an application in order to execute our tool, the tool is meant for Palm application developers. The checkpointing mechanism is not automated because we would like the checkpointing APIs be utilized in a mobile computing application, and then a proper checkpointing and rollback recovery protocol determines the right time for checkpoint taking or recovery. A mobile computing application using Palm OS devices can be a joint statistical computation between several mobile devices equipped with a wireless network interface.

In Section 4.3, we measured the execution time when the effective checkpoint size is as large as 3 Mbytes. For a real Palm application, its dynamic data cannot be that large because the dynamic heap is only 256 Kbytes. It is expected that, with newer versions of Palm OS, the limitation will be lifted when the physical memory size of a Palm OS device is also enlarged.

5. Conclusions

In this paper, we described the importance of assuring the reliability of a distributed computing system. Since a mobile/handheld device suffers from limited vulnerable storage and power limitations easily, it is essential to keep the computing state when a mobile/handheld device is participating in a dependable distributed computation. Once a failure occurs, the device can recover from its last checkpoint, and then resume computation. For this reason, we developed a generic checkpointing tool for the most popular operating system – Palm OS, among handheld devices. With the checkpointing tool, an application can take a snapshot of its executing state, save it as a record in the storage heap, and then further saved onto a PC by taking a synchronization with the built-in HotSync tool. Or, with the IrDA functionality on the handheld, we can transmit the checkpoint to another handheld, and then restore execution on that handheld.

To the best of our knowledge, our tool is the first checkpoint and recovery tool in Palm OS. We applied the checkpointing tool to four popular Palm applications. The result showed that our tool is very useful. Because of its generality, we believe that our checkpointing tool can be applied to many other Palm applications and will become a popular tool to be used in dependable mobile computing research.

6. References

[1] E. A. Brewer, et al., "A Network Architecture for Heterogeneous Mobile Computing," *IEEE Personal Communications*, Vol. 5, No. 2, pp. 8-24, 1998.

[2] IDC, *Personal Companion Market*, 1999.

[3] Palm Inc., "Chapter 6, Memory," *Palm OS Programmer's Companion*, 2000.

[4] Neil Rhodes and Julie McKeehan, "Chapter 2, Development Environments and Languages," *Palm Programming – The Developer's Guide*, 1st Ed., O'Reilly, 1999.

[5] Palm Inc., "Chapter 4, Event Loop," *Palm OS Programmer's Companion*, 2000.

[6] Palm Inc., "Chapter 7, Files and Databases," *Palm OS Programmer's Companion*, 2000.

[7] Palm Inc., "Chapter 8, Palm System Features," *Palm OS Programmer's Companion*, 2000.

[8] B. R. Badrinath, A. Acharya, and T. Imielinski, "Impact of Mobility on Distributed Computations," *SIGOPS Review*, Vol. 27, No. 2, pp. 15-20, 1993.

[9] E. Elnozahy, L. Alvisi, Y.M. Wang and D.B. Johnson, "A Survey of Rollback Recovery Protocols in Message Passing Systems," *Technical Report CMU-CS-99-148 School of Computer Science, Carnegie Mellon University*, June 1999.

[10] R. Prakash and M. Singhal, "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 10, pp. 1035-1048, 1996.

[11] N. Neves, W. Kent Fuchs, "Adaptive Recovery for Mobile Environments," *Communications of the ACM*, Vol. 40, No. 1, pp 68-74, January 1997.

[12] K. F. Ssu, B. Yau, W. K. Fuchs, N. Neves, "Adaptive Checkpointing with Storage Management for Mobile Environments," *IEEE Transactions on Reliability*, Vol. 48, No. 4, pp 315-324, December 1999.

[13] Palm Inc., "Chapter 3, Application Startup and Stop," *Palm OS Programmer's Companion*, 2000.

[14] Palm Inc., *Palm OS SDK Reference*, 2000.

[15] "Palm OS SDK version 3.5," http://www.palmos.com/dev/tech/tools/sdk35.cgi

Acknowledgement

The authors would like to thank Chung-Yi Wang for his helpful discussions.