Hoarding for a Hierarchical Storage Architecture

Christopher LaRosa

May 2003

Computer Science Department
Hamilton College
198 College Hill Road
Clinton, NY 13323

Abstract

Despite recent increases in the processing power of handheld computers, often called palmtops, the devices continue to function primarily as personal information organizers. We believe adding mass storage to handheld computers will expand the usefulness of the devices. We propose a storage hierarchy, involving a hard disk and large software controlled flash memory cache, that lowers the power cost of accessing the data from the disk, but retains the storage capacity benefits of the disk. We investigate how the hoarding of files into flash memory during a power abundant docked state can drastically reduce the power consumption of the hard disk.

1 – Introduction

During the past half-decade handheld computers have become fixtures of mainstream portable computing. Following failed attempts by several companies during the early 1990s to popularize handheld computers, Palm Computing's development efforts led to wide adoption of its PalmPilot handheld computers during the latter half of the decade. Today numerous hardware vendors sell handheld computers running PalmOS, Microsoft Windows CE, and embedded versions of Linux. Historically, handheld computing capabilities have paled in comparison to desktop and laptop capabilities. Slow processors, limited memory, a lack of mass storage, slow and costly network connectivity, and awkward user-input peripherals have meant that handheld computers have never been widely used beyond the scope of their original purpose — personal organizers. When handheld computers are used in conjunction with a desktop application, such as the Palm Desktop software, handheld computers are often further limited to function primarily as information retrieval devices.

Recently, the difference between the hardware powering handheld computers and the hardware powering desktop computers has significantly narrowed. The hardware components of both a modern handheld computer, the HP iPaq, and a three-year-old laptop computer, the Apple PowerBook G3, include a 400 megahertz processor, 64 megabytes of RAM, and a color display. Despite the similarities between handheld and laptop platforms, the handheld computer is rarely used for the day-to-day business applications such as word processors, email clients, and spreadsheets that the three-year-old laptop computer is regularly used for. Their continued lack of fast input devices,

large displays and mass storage prevent handheld computers from moving beyond their stagnant role as personal information organizers.

Overcoming the display, input, and storage constraints of palmtops would mean that mainstream users could effectively replace all of their computers – desktops, laptops, and handhelds – with a single device capable of running all of their applications. With a full-function handheld computer available, the desktop and laptop could be removed altogether from the computing paradigm. The benefits of a using single handheld computer in place of multiple computers are many: lower cost of hardware and software, increased mobility for the user, decreased maintenance needs, consistency of data in all locations, and increased availability of user applications in mobile settings. Overcoming the display and input deficiencies of handheld computers using traditional display and input devices would essentially turn handhelds into laptops without hard disks. For most users though the usefulness of constant data availability surpasses the need for robust input and display capabilities. This is demonstrated by the popularity of the Palm handheld/Palm Desktop solution in which users do the bulk of their data entry and editing on the desktop and regularly access their data in all locations.

Users needing constant data availability would benefit from the single handheld device paradigm if the handheld device were extensible with input, output, and display peripherals via a dock. With a dockable handheld that provides mass storage capability a user would take a single handheld computer everywhere. While at work, at home, or at a public terminal the user could dock the device into a full-sized display, keyboard, mouse, and a high-speed network or printer if necessary. When away from docking stations the user would benefit from being able to access all of his or her documents and have full use

of all applications at all times. With current palmtop hardware and the addition of an I/O docking station, the only component necessary to realize the single handheld computer paradigm is the addition of mass storage to the handheld computer.

Mass storage in a handheld computer must meet different criteria than mass storage on a desktop or laptop computer. Of utmost concern in design of handheld mass storage is power consumption. While hard disks, as used in desktops and laptops, provide the storage capacity needed by handheld users, they consume far too much energy for a handheld device. A laptop battery, typically the size of a handheld computer, spends much of its energy powering a laptop's hard disk. Non-volatile flash memory, traditionally used for primary storage in handhelds, is too expensive to provide sufficient storage for most users' document and application collections. Thus, in order achieve the ideal handheld design in which users have sufficient storage for the entirety of their data, a solution that provides low-energy access to all of a user's data should be investigated. In much the same way a memory hierarchy minimizes the time cost of memory access, providing fast access to commonly used data while maintaining the ability to access all data at a lower speed, some sort of hardware hierarchy could be devised to minimize the energy cost associated with mass storage. A logical starting point for computer scientists to address this problem is to use new software methodologies with existing hardware technologies to take full advantage of the functionality provided by power-hungry devices while minimizing the devices' impact on battery life.

2 – Related Work

Researching a way to efficiently provide data from large collections of user and program files should begin at the intersection of previous bodies of investigation focusing on data availability and power aware software in mobile computers. Research aimed at minimizing power use in mobile systems through system software design is directly related to the task of minimizing power consumption for storage access. In addition, research that focuses on providing files to laptop users, especially in computing milieux where data access is historically constrained by the limitations of weakly connected, low-throughput, or power expensive wireless networks should be relevant to the task of enhancing file availability on the handheld.

Existing research on overcoming wireless networks' shortcomings investigates different methods of preemptively moving files from a server, intermittently inaccessible due to network unavailability, onto a mobile device to ensure data availability. This process of intelligent predictive caching is called *hoarding*. File systems and data replication programs have demonstrated that quick seamless availability of files stored on an intermittently connected server is possible using a variety of algorithms for hoarding: least recently used hoarding, user directed hoarding, frequency based hoarding, and predictive algorithms.

2.1 Coda – A hoarding file system

In 1987, researchers at Carnegie Mellon University began constructing a network file system named Coda that replicated server data on client laptops to ensure data availability during periods of network disconnect [Sat02]. Coda — now 15 years in development and included in Linux distributions — is a logical foundation for new

hoarding research. The Coda research group pioneered an *optimistic replication* strategy for data replica control. Optimistic replication was a rejection of previous replica control algorithms that limited data replication of shared files to ensure that in shared files would always be consistent among different users. Coda researchers found that limiting data replication in favor of consistency put too many limits on data availability. In considering the findings of Coda's researchers, it is important to note that a contained hoarding file system, one in which only a single user has access to replicated data, consistency of data between users is a non-issue. None of drawbacks that taint the utility of optimistic replication in the multi-user file server environment that Coda is used in exist for single-user systems. Coda's caching subsystem, Venus, achieved optimistic replication via a three state hoarding process: *hoarding*, *emulating*, and *reintegrating*. During Coda's hoarding state, Venus, though a combination of Least Recently Used (LRU) based algorithms and user-defined file sets, determines which files should be copied from a server onto a client laptop for use during disconnection from the server. Coda's emulating state begins when the laptop is disconnected from a server and ends when the laptop is reconnected to a server. During its emulating state, Coda maintains a *client modification log* that tracks changes to cached files and tracks cache misses. During its *reintegration* with a server Coda replaces files on the server that were modified during emulation, comparing files that were changed on both the server and the laptop and prompting the user for guidance in file replacement. Again, in the single user handheld environment files are created and edited by only one user. Thus, this important drawback found in multi-user environments is a non-issue.

2.2 – Frequency Based File Caches

In 1990 Robinson and Devarakonda described a modification to the traditional LRU strategy used in file caches [Rob90], including Coda's Venus system. They observed that temporal locality plays less of a role in file caches than in CPU and memory caches. Robinson and Devarakonda's new Frequency-Based Replacement (FBR) algorithm addressed the fact that certain data items that are accessed repeatedly over longer intervals of time expire in LRU caches despite the high utility of maintaining such items in the cache. The FBR designers were able to improve on the effectiveness of LRU data caches by restructuring the cache into three temporal zones: new, middle, and old. On a cache miss, a file is brought into the new section of the cache which behaves exactly as an LRU cache does. When an item expires from the new zone, it moves to the old zone where it is given an integer reference count. The old zone behaves exactly like a traditional frequency based cache where the least-referenced files expire first. When a cache hit occurs on a file in the old zone it is moved to the middle zone where it retains its reference count. With each subsequent reference the file's reference count is increased. As the middle zone fills, expired files move back into the old zone where they replace those files in the old zone with the lowest reference count. The old zone, being fed by both the middle and new zone, serves the important task of catching repeatedly referenced items that are about to expire from the cache so they can be inserted into the middle section. Throughout the middle and old zones reference counts are regularly reduced using ceiling division by two whenever the cumulative total of reference count reaches a certain threshold. This keeps files with artificially high reference counts — the result of rapidly repeated accesses while a file is occupying middle section — from

becoming fixed in the cache. The effect of this three-zone data cache architecture is that accessing files that are repeatedly accessed for a short period of time result in cache hits (from the new zone) and accessing files that are repeatedly accessed over long periods of time result in cache hits (from the middle zone). Across five test systems running three different operating systems the FBR algorithm showed improvement over LRU algorithm in cache hits in all but two of twenty-five tests. Cache sizes were varied from 800K to 32MB. At 8 MB the FBR algorithm showed significant relative improvement over the LRU algorithm by approximately 29% (this means that cache misses occurred 3-4% less often). The FBR algorithm's cache hit rate approached that of the LRU algorithm for larger caches in a UNIX environment. The FBR algorithm makes logical sense for hoarding since users commonly work on individual files for short periods of time, such as papers and letters, and access other files continuously, such as system files and application files.

2.3 – SEER: Project-based file hoarding.

In 1997 Kuenning and Popek developed a file hoarding system called SEER that based its hoarding algorithm on the fact that users generally access distinct groups of files when they are working on different projects [Kue97]. SEER groups files into project clusters and decides which clusters to hoard rather than deciding which files to hoard individually. The SEER research group defines the metric of *semantic distance* as the likelihood that access to a file will be closely followed by access to a different file and used the metric to build their hoard clusters. The SEER group makes use of three different ways to calculate semantic difference: *temporal semantic distance* (the time between file access), *sequenced-based semantic distance* (the number of intervening file

access between two different files), and *lifetime semantic distance* (if two files are open at the same time the distance is zero, otherwise the distance is the sequence-based value). SEER ignores semantic distances for files not accessed within 100 sequential file accesses of each other, capping the complexity of its hoarding algorithm. SEER clusters files that have a short semantic distance to a common file into projects. SEER's research group considered several implementation problems including accounting for meaningless activity (e.g., execution of the `find` command) and how to treat temporarily created used and deleted directories and files. Kuenning and Popek showed that when using SEER's cluster based algorithm, a cache size only slightly larger than the size of a user's working data set was necessary to ensure miss-free operation of the cache, whereas when using LRU based caching algorithms more than twice as much space as the working file set was needed to ensure miss-free operation. In SEER the hoard file selection (cluster forming) algorithm is processor intensive, comparing each file access with its closest 100 accessing. SEER's operation while disconnected from the server incurs minimal processor overhead.

2.4 – Tree based hoarding algorithms.

In 1995 Tait *et al* developed and implemented a tree-based hoarding algorithm where each program's execution produced a file trace tree [Tai95]. In their approach, implemented under OS/2, tree nodes contained process IDs, their children contained subtask process IDs, and their leaves contained the paths of files that have been accessed by their parent process. The tree-based approach requires little user interaction. In the background the system builds hoard trees for different programs and then prompts the user prior to disconnection to specify which programs will be run during disconnection.

In place of the user prompt, the system can also select programs automatically using an LRU algorithm. Experimentation showed that tracing data adds 2% processor overhead to file access. Tait's group reached limited conclusions about the effectiveness of their algorithm because they never had large sets of real world data to work with. While their approach has not been proven effective, it forms a parallel to SEER's file association algorithm, only the tree-based algorithm tackles the easier task of examining the operating systems file associations rather than determining user's file associations.

2.4 – Power-aware system software

Existing research that has focused on system software level modifications designed to improve performance and decrease energy usage is also applicable when considering how to overcome the drawbacks of energy hungry components. Researchers at Purdue and UCLA have shown that transferring processor tasks from energy constrained mobile devices to energy abundant servers can save significant power on the handheld [Ru98, Li01]. This body of research is heavily focused on establishing heuristics that are used to determine under what combinations of network and processor variables energy savings can be wrought by offloading processing.

In 2001 the Purdue group published findings that outlined a partition scheme for joint program execution across energy constrained mobile devices and energy abundant servers. Their experimentation demonstrated that moving processor intensive tasks off handhelds often resulted in energy savings and, not surprisingly, that program execution is accelerated when moved to the faster server. The Purdue group also found that mirroring of data on both the server and the client minimized traffic between the two, saving energy and that potential performance improvements were possible by

overlapping computation with communication. This overlap rule said that executing a task, whether or not it actually needs to be executed, during idle time on the energy abundant server, saves time and energy if the energy constrained handheld should eventually need the task to be completed. If the task is already complete the handheld spends idle time waiting to receive the result of the computation.

3 - Hypothesis

Previous research indicates that it is possible to accurately predict what files users will use. This accuracy is demonstrated by the success of SEER's hoarding algorithm, and the real world usage of the Coda file system. Hoarding and pre-fetching research shows that, although users have several hundred megabytes, sometimes gigabytes, of documents, they typically will use only a small fraction of their data during any work session. This research also shows that it is possible to predict which files will be accessed. Research on processor cycle offloading shows that completing tasks, even unnecessary tasks, in an energy abundant-environment saves power and increases performance on mobile computers. Considering these two findings together, it follows that a caching storage hierarchy can be built to lower the power access costs normally associated with disk-based storage systems. Using a hoarding file system that fills a power-efficient flash memory cache with files from power-costly hard disk during the handheld computer's energy abundant docked state is a possible solution to meeting the high-capacity, low-power storage needs of handheld computers, and facilitates the arrival of the single-computer handheld paradigm. The software-controlled hoarding in this hierarchy would take advantage of the energy abundant docked state to perform costly access to the disk, copying files into low-power flash memory that the user is likely to

use later when the handheld computer is untethered from the dock. The feasibility of the hardware design, software implementation, and impact on battery life of the new storage hierarchy we have described should be investigated. We investigate the effectiveness of hoarding for hierarchical storage, simulating a system where storage hardware consists of a flash memory cache and a hard disk, and determine the impact of the new storage architecture on handheld's battery life, and whether *hoarding, hierarchical storage (HHS)* should be investigated further.

Before this storage architecture is given further consideration, the physical impact of adding a hard disk and flash memory to a handheld design must be determined to be feasible. The specifications of current mobile components show that the addition of a hard disk and flash memory cache would not drastically alter the shape or size of handheld computers like the Hewlett Packard iPaq or Sharp Zaurus. Toshiba manufactures a 2.12 inch wide, 3.09 inch tall, and .2 inch thick 10GB hard disk, the HDD1262, that would meet most users' storage demands and meet handheld computer size requirements. Handheld computer designers might find that many users would trade features such as biometric finger-print readers, CMOS digital cameras, and GPS positioners currently consuming space on handheld computers for ten gigabytes of storage. Already handheld computers ship with flash RAM as the primary storage medium, increasing the capacity to accommodate a hoarding cache would not alter the memory architecture. Also, the HDDD1262 operates at the same voltage as flash memory and comes bundled with PCMCIA interfaces, so handheld power architectures and peripheral busses would not need revision to accommodate the disk drive. Given the

physical appropriateness of the solution, an investigation of hoarding's impact on power consumption is in order.

Traditionally, handheld computers store all of their system, application, and user data in flash memory. Flash memory provides low-latency, low-power storage. If the single-handheld computer is to replace users' desktops and laptops, storing all system, application, and user data in flash memory is no longer possible. Adding a hard disk to the storage system raises the question of whether system and application data should be stored in flash memory by default. Since all commonly used files will eventually reside in the hoard cache, it makes sense to move the entire operating system and all application data out of flash memory where unused application and system files will consume valuable memory that could be utilized for the hoard cache. When a user carries data that occupies less space than their cache's capacity, their hard disk will never need to spin up. In this case, the hoarding software can put all files into the cache by default. Thus, we want to develop a model for the impact on battery runtime resulting from the introduction of HHS for cases where users' storage needs exceed the amount of available flash memory.

5.1 – A power cost model

To model the impact on battery life that results from adding HHS, it is first necessary to model the additional power consumption introduced by the new storage architecture. A power model of HHS accounts for access to two sets of files: *h*, files hoarded in flash memory, and *n*, files stored only on the disk. The cost of accessing a hoarded file *i* is the product of the power needed to access the file from flash memory

(F_i), and the number of accesses to that file during a session (A_i). The total cost of access to all hoarded files is $C_h = \sum_{i}^{h} F_i \cdot A_i$.

We model two cases for accessing files off the disk. In the first case the handheld device is rendered useless while the disk is spinning up, since either an executing application or the user has requested necessary data and will likely not move on until the data is available. Well call this case disruptive. We also model cases where user and application progress continues during spin up. We call this case, where progress continues, non-disruptive. When modeling the power cost of accessing files from the disk, it is necessary to account the cost incurred by reading the files from the storage medium (D_h), as was the case in our hoarded file power model. We also account for the power use associated with spinning up a sleeping disk when a file is needed. The power associated with each spin up S includes both the power cost of spinning up the disk (S_p), and, in the case of disruptive access, the overhead of powering the rest of the handheld during the spin up time (S_t). It is also a safe assumption that, when using handhelds untethered from docks, users have in mind a defined task and waiting for file access, as occurs in our disruptive module, prolongs the time necessary for them to complete the task at hand. Handheld overhead (O_h) is the power necessary to maintain the display, operate the processor, and refresh memory while the handheld is idle waiting for the hard disk to spin up (S_t) during the disruptive model. Handheld overhead should be measured in power per second, so that power cost can be expressed as a product of spin up overhead, O_h, and disk spin up time, S_t. The number of spin ups necessary during a session is related to the number of access to files that are not hoarded. It makes sense to

cache each non-hoarded file on its first access since the access cost from the disk is so high and the file has been determined to be needed by the user. Because access to non-hoarded files may be clustered, each access may not incur a spin up. For lack of an explicit formula expressing the number of spin ups necessary for access to non-hoarded expressed, we use the unknown function $S(n)$ to express the quantity. The total cost of spin ups is $S_t[O_n + S_c] \cdot S(n)$. Also, we consider that after each file access we will let the disk spin idle for some short time t — the idle spin threshold — in case another file needs to be opened. This idle time too is unknown, we define it as a function of the file set accessed and the idle spin threshold $I(n,t)$. The power cost associated with this idle time we express as $D_{idle} * I(n,t)$ where D_{idle} is the cost of idling the disk. The total cost of access to non-hoarded files is the cost of accessing each non-hoarded file once plus the cost of accessing the file from flash memory for each access other than the initial access

$$\left[\sum_{i}^{n} D_i + S_t[O_h + S_c] \cdot S(n) + D_{idle} * I(n,t)\right] + \left[\sum_{i}^{N} F_i \cdot (A_i - 1)\right].$$

With the cost of both hoarded and non-hoarded access defined, the total power cost of file access in a hierarchical hoarding storage architecture is the sum of the total energy cost of accessing hoarded files, and the cost of accessing non-hoarded files:

$$C_{hhsa} = \left[\sum_{i}^{h} F_i \cdot A_i\right] + \left[\left[\sum_{i}^{n} D_i + S_t[O_h + S_c] \cdot S(n) + D_{idle} * I(n,t)\right] + \left[\sum_{i}^{n} F_i \cdot (A_i - 1)\right]\right].$$ An

optimal hoard-based memory hierarchy would consume no more power than a traditional flash memory-only handheld storage architecture for similar file access patterns. The cost of accessing all files in *h* and *j* from flash memory is our basis for comparison:

$C_{optimal} = \left[\sum_{i}^{h} F_i \cdot A_i + \sum_{i}^{n} F_i \cdot A_i\right]$. The change resulting from the addition of a hoard-based hierarchical file system is the difference in cost of HHS and optimal model:

$C_{hhsa} - C_{optimal} = \left[\sum_{i}^{n} (D_i \cdot F_i) + S_t[O_h + S_c] \cdot S(n) + D_{idle} * I(n,t)\right]$. Since the cost of access off the disk is less than the access cost of the flash, we assume that whenever the spin up cost is significantly larger than the difference in read cost, as it always is with disks, the power cost will be negatively affected by an increase in the number of access to non-hoarded files.

To simplify our model for experimentation we acknowledge that there is an average file size (*ave*), and hence an average cost of access from the flash memory (F_{ave}) and an average cost of access from the disk (D_{ave}).We determine *S(n)* and *I(n,t)*'s values experimentally. For simplicity sake we redefine the power cost difference using the total number of non-hoarded files, $q = |n|$:

$C_{diff} = q \cdot [(D_{ave} - F_{ave})] + S_t[O_h + S_c]S(n) + D_{idle} * I(n,t)$ We also analyze our experiments assigning an overhead (O_h) value of zero to show a simulation where file access do not obstruct user progress. Real world performance will lie somewhere in between the two sets of results.

5.2 – A battery impact model

For our experimentation to have real meaning we would like to know what change in battery life we can expect as a result of the addition of HHS. To determine the change we define the runtime of the optimal device, the handheld computer's battery runtime without any hardware modification, and the runtime of the handheld with the

addition of HHS: $R_{orig} = \frac{batterycapacity(watthours)}{averagedraw(watts)}$; $R_{hhsa} = \frac{batterycapacity(watthours)}{averagedraw(watts)+C_{diff}}$. The runtime of the palmtop with HHS expressed as a percentage of original runtime is $R_{\%} = \frac{R_{hhsa}}{R_{orig}} = \frac{averagedraw(watts)}{averagedraw(watts)+C_{diff}}$. Also, we consider the impact of running a hard disk continuously and calculate the improvement of our battery runtime over the continuous hard disk model to see what percentage of power is recuperated by hoarding for hierarchical storage.

4 – Methods

To test our hypothesis that adding hoarding system for a hierarchical storage architecture to a handheld computer would provide mass storage with minimal impact on battery runtime, we simulated the operation of HHS using real-world file usage patterns. We collected file usage traces and, using a simple frequency based hoarding algorithm, manually simulated the operation of HHS to determine its impact on battery life.

4.1 - Experimentation Platform

We chose the Linux platform as the basis for our investigation. We favored Linux over the most common handheld operating systems, Windows CE and PalmOS, because Linux provides greater ability to customize file system functions thereby enables easier prototyping of a HHS during later research. We also favored Linux over traditional handheld operating systems because, as a more full-featured operating system, Linux provides the capabilities necessary for users to replace their desktop and laptop computers with handheld computers, advancing the single-device paradigm we described earlier.

Two major hardware lines of handheld computers run Linux: the Sharp Zaurus series, which comes with an embedded version of Linux called Embedix installed, and the Hewlett Packard iPaq series, which comes Microsoft Windows CE installed but is capable of running Linux. We used the Debian distribution of Linux for our experimentation because it can be compiled to run both on the Zaurus and on a desktop machine where much of our simulation takes place.

For our experimentation hardware we used a Hewlett Packard Vectra VL with a 166 megahertz Pentium processor and 52 megabytes of memory. Our test system's processor and memory capabilities are at or below those of current iPaqs and Zaurus which have 200-400 megahertz processors and, usually, 48-64 megabytes of memory.

4.2 - File Usage Traces

The most significant unknown in our HHS paradigm is how having a full-featured handheld computer with mass storage will change users file usage patterns. This makes the task of defining a representative work session and its associated file trace very difficult. Even in traditional laptop and desktop settings, gathering file access traces that are representative of users' real world working habits has been a stumbling block in previous hoarding research. Furthermore, much of successful file-hoarding research has been undertaken on OS/2, DOS, and non-BSD Macintosh systems where research methods do not relate to our Linux environment. The Storage System Lab at Hewlett Packard Research provides traces of three months of disk access on UNIX workstations circa the early1990s. The HP UNIX disk traces, commonly used in file system and disk research, provided data about usage at the block level; we found the inability to relate the data with an actual system a major drawback. For our investigation we wanted to know

specifically which files do not enter the hoard cache so that we can suggest future work to improve file hoarding. During our experimentation we gathered our own trace data from different sessions on our test hardware.

We investigated two approaches to file-tracing simultaneously: running the Linux Trace Toolkit, an open source general purpose kernel tracing package for LINUX partially developed by IBM, and rewriting the *open_sys* functional call in the kernel source to log file opens.

4.1 – The Linux Trace Toolkit

The Linux Trace Toolkit (LTT) consists of a kernel patch, a trace module, a daemon, and a standalone analysis application [Yag00]. The kernel patch adds mechanisms throughout kernel functions that communicate log events to the trace module. The trace module collects log information into a buffer and the daemon periodically empties the buffer into a log file. The trace module's buffer serves the important purpose of minimizing overhead on the system. The analyzer converts the daemon's output file into readable ASCII text or displays the content of the trace on the screen. Unfortunately the amount of data provided by the LTT is overkill for the needs of a hoarding algorithm. Approaching 6 megabytes of file system log data per minute, much of the data collected is merely the file system extending the timeouts of already open files and the module logging its own activity. The analyzer is very slow, taking days to convert output files into meaningful logs. Moreover the LTT only provides the name of accessed files, not their paths. With many duplicate file names on a system this too is an inadequacy. Using Perl and Linux's locate command we wrote a program that could automatically match unique file names to file paths, prompt the user in the case of

duplicate file names, and recognize patters in users selection to minimize the redundant input demands on the users. This system, although cumbersome, provided us with a list of files opened during a given trace session, the number of access to each file during a trace session, and a basis to test our hoarding algorithm.

4.2 – Modifying *sys_open()*

In an attempt to avoid the workarounds necessary to gather useful traces from the LTT we also investigated what modifications to the LINUX kernel would be needed to log file calls to *sys_open()*. Because C's buffered I/O capabilities are not available from within the kernel we determined that inlining functionality into *sys_open* would be the only simple way to record file access from within the kernel. In this unbuffered I/O scenario the cost of file opens on performance would be doubled. Each file open would result in a write to the log file. Also, by the time we determined inlining was the appropriate route to achieve our goal, the functionality our Perl program supporting the LTT had matured to a point where it could develop an accurate file trace with relative ease and we put our attempts at logging though custom kernel modification on hold.

4.3 – Defining the trace environment

We collected five user traces in total. We recorded file access during two 15-minute periods that we believe accurately represent a session when a user would be using their handheld in docked mode to perform significant editing on word processing documents, spreadsheets, and email messages. We also record file access during three two-minute sessions we believe accurately represent a session in which the user, operating the handheld in a mobile setting, retrieves information from a few documents, checks their email, and briefly edits documents. During our trace, OpenOffice, a

MicroSoft Office-like productivity bundle, was used for word processing and spreadsheets; Mozilla was used for email and web browsing.

4.4 - Measuring Hoard Performance

We conducted a series of frequency based hoard simulations to measure performance. Using a custom program called the Hoard List Generator, a Perl program of approximately 175 functional lines, and a combination Linux's *sort* and *locate* utilities to analyze a file trace log we established: the average file size accessed during the session (*ave*), the total number of files accessed during the session, and a hoard list containing the paths of files accessed during the session in descending order of frequency. For a flash memory cache of size N bytes, we put the first N/*ave* files of the hoard list into the simulated hoard cache. We then manually simulated what would happen when another trace session was run on HHS, observing which files were accessible from the simulated hoard cache, how many seconds of idle spin time would result, and how many spin ups would be needed. We then put the miss count into our cost difference equation from Section 3 along with sample power information from a modern mobile hard disk. Using the energy specifications of a modern palmtop computer and the power cost difference equation we determine the impact on battery runtime HHS incurs.

5 – Results

To investigate the effectiveness of a simple Frequency Based Hoarding algorithm in HHS we collected a series of traces. We collected information about the total number of files accessed during a session, the total size of files accessed during a session, the average accessed-file size during a session, and the average file access interval, the

average number of seconds between file opens involving the disk during session. Below are our findings for two 15 minute traces and three 2 minutes traces.

5.1 – File trace findings

trace name	total files	total data size	average file size	ave. access interval
15 minute a	504	111.6 MB	226.7 KB	1.8
15 minute b	502	114.2 MB	232.9 KB	1.8
2 minute a	182	87.7 MB	493.4 KB	.7
2 minute b	43	11.1 MB	264.3 KB	2.8
2 minute c	45	6.9 MB	157.0 KB	2.7

Fig 5.1 – Trace Statistics

Because our hoarding simulation uses average file size to calculate the number of files that can be stored in the hoard cache, we computed the average file size using all five disk traces as input to be approximately 250 kilobytes. This average, accounting for over a half an hour of use, is more representative of the actual average size than the 15 minute trace averages and especially the widely varying 2 minute trace averages.

It should also be noted that the total data size does not necessarily represent the amount of data read during a session. It represents the summation of the sizes of the files that were opened. Applications can open a large file and only read a portion of the file's data. However for our hoarding algorithm we will only consider caching full files to simplify our experimentation.

5.2 - Hoard Simulation

In our first test we decided to use one fifteen-minute file trace as input to the Hoard List Generator. This means only file accesses during the *15 minute a* trace were used to construct the frequency based hoard cache. We simulated the cache's operation for each of the two-minute trace sessions with 64 megabyte and 96 megabyte caches. We choose these sizes for our memory cache because our trace data was limited and we

wanted to test the robustness of the hoarding algorithm. This cache size also seemed a logical test of the hoarding algorithm because the larger cache size had sufficient capacity to store all of the data that was used during each of the two-minute sessions, which simulated mobile use.

trace	250 file cache ≈ 64 MB Cache				400 file cache ≈ 96 MB			
	hits	misses	hit rate	interval	hits	misses	hit rate	interval
2 min. a	61	121	.34	.99	110	72	.60	1.66
2 min. b	22	21	.51	5.74	28	15	.65	8.00
2 min. c	21	24	.47	5.00	26	19	.57	6.31

Fig 5.2 – Simulation results with *15 minute a* as Hoard List Generator input

Although we recorded hit rates as high as 65%, we noted the average file access interval, the time between opens requiring disk access, was still near or below the spin up time, meaning disk will remain spinning almost constantly, consuming power and leaving us with little chance for improvement over the baseline measurement of battery life of a handheld with non-hoarding disk storage. We increased the amount of trace data into the Hoard List Generator to include the traces *15 minute a*, *15 minute b*, and *2 minute a*, and simulated the cache's operation for *2 minute b* and *2 minute c*.

trace name	250 file cache ≈ 64 MB Cache				400 file cache ≈ 96 MB			
	hits	misses	hit rate	interval	hits	misses	hit rate	interval
2 min. b	29	14	.67	8.57	34	9	.79	13.30
2 min. c	30	15	.67	8.00	36	9	.80	13.30

Fig 5.2 – Simulation results with multiple traces as Hoard List Generator input.

While the hit rate did improve it did not reach the level of success we would have liked to have seen given the consistency of the user behavior during our different traces. Inspecting the hoard misses we found two situations where the frequency based hoard generator resulted in misses.

The primary cause of misses while simulating *2 minute c* was OpenOffice accessing a collection of templates and fonts that had not previously been accessed. This failure might work itself out as more trace data becomes available for the frequency based hoarding algorithm, or the failure could be addressed by introducing a program-trace based hoarding algorithm like SEER. In *2 minute b* misses occurred when Mozilla attempted to make use of its own file cache. It is reasonable to assume that for any reasonably fast network, which is already connected if users are accessing web content, the cost of transferring four files across the network will be less than spinning up a hard disk anywhere from one four times depending on the density of the cache accesses. We re-ran the simulation as if Mozilla's file cache was disabled using *15 minute a*, *15 minute b*, and *2 minute a* as inputs to the Hoard List Generator and obtained the following results:

trace name	250 file cache □ 64 MB Cache				400 file cache □ 96 MB			
	hits	miss	hit rate	interval	hits	miss	hit rate	interval
2 min. b	29	8	.78	15	34	3	.91	40.00
2 min. c	30	15	.67	8.00	36	9	.80	13.30

Fig 5.3 – Simulation results with multiple traces as Hoard List Generator input and no Mozilla file cache.

With the 96 MB cache hit rates averaged 80% and 91%. We used the following hardware power specifications for a Toshiba HDD1262, Lexar flash memory card and Sharp Zaurus SL-5600 (appendix a) as parameters in our models for the case when the Hoard List Generator was given three trace files for input and Mozilla's cache was ignored. We calculated the percentage of battery life for disruptive access and non-disruptive access as a percentage of the unmodified Zaurus, and as a percentage of runtime for a Zaurus with a non-hoarding hard disk. The baseline comparisons were 111 minutes runtime for the non-modified Zarus and 85% of original runtime for the non-hoarding hard disk model.

trace name	250 file cache ≈ 64 MB Cache, 5/10 second spin down threshold						
	miss	*I(n,t)* idle spin time	*S(t)* # spin ups	% runtime orig. - disruptive	% runtime orig - non-disruptive	% runtime continuous disk - disruptive	% runtime continuons disk - non-disruptive
2 min. b	8	22/42	4/4	.84/.82	.93/.91	.99/.96	1.10/1.07
2 min c	15	43/61	6/3	.77/.83	.89/.89	.91/.97	1.05/1.05

trace name	400 file cache ≈ 96 MB Cache, 5/10 second spin down threshold						
	miss	*I(n,t)* idle spin time	*S(t)* # spin ups	% runtime orig - disruptive	% runtime orig. - non-disruptive	% runtime continuous disk - disruptive	% runtime continuons disk - non-disruptive
2 min. b	3	10/20	2/2	.91/.90	.97/.95	1.08/1.06	1.13/1.12
2 min c	9	27/42	3/3	.86/.85	.93/.92	1.02/1.10	1.10/1.08

Figure 5.5 – Power consumption changes

Our findings showed that for cases where the hoarding success rate was less than 80%, both of the 64 megabyte traces with both 5 and 10 second spin down thresholds, power was lost due to spin ups over the baseline of running the disk constantly where access was disruptive. Where access was not disruptive battery runtimes were 89%-93% of the original runtimes and showed 5-9% improvement over non-hoarded disk storage. For a 96 megabyte cache, where all hit rates were 80% or better, we saw runtimes of 85-91 % of original runtimes with our disruptive model and 92-97% of original run times with our non-disruptive model. Compared with the non-hoarded disk based storage, our hoarding algorithm with a 96 megabyte cache showed up to 13% runtime improvement.

6 - Conclusions

With our simple frequency-based hoarding algorithm and a 96 megabyte cache we were able to simulate runtimes ranging from 85%-97% of a non-modified handheld. This 3%-15% reduction in battery runtime is far less than that which is created by wireless networking devices which often halve battery life. Given the utility of added storage we believe that adding HHS with a sufficiently large cache is a promising

solution to providing mass storage to handheld devices. Also, when coupled with better hoarding algorithms we would expect far less severe impacts on battery life.

7 - Future Work

Future research should be focused on improving hoarding in a handheld environment so that the hoard hit rate can approach 100%. In our limited experience with hoarding simulations increasing the amount of history greatly improved the effectiveness of the frequency based hoard algorithm. Doubling the amount of historical data nearly doubled the effectiveness of the frequency based cache. Research should be conducted to determine at what point adding trace data no longer improves hoarding. Experimentation with existing hoarding algorithms, especially process based hoarding algorithms, seems promising as hoard-misses often arrived in groups when an application performed a task it had not done during any trace histories. Also, determining an optimal idle spin threshold should further improve the effectiveness of HHS. For further accuracy, measuring the true impact of file tracing on battery life should be investigated to produce more accurate models for the power cost incurred by HHS.

8 - Acknowledgements

I send my thanks to Mark Bailey, my thesis advisor, for his guidance though out this project and extend my gratitude to the entire Computer Science department faculty whose tutelage over the past four years prepared me for this investigation. I also would like to thank Julie Parent for her always being happy to share her advice and knowledge with me. Finally I extend my thanks to the students at the Writing Center who have helped in revising sections of this document.

9 – Appendices

a – medium power cost table

	Lexar flash memory	Toshiba MK1003GAL
power/MB read	5.52*10^-5 W	2.34*10^-5
wake time	<10 mSec	3.5 sec
system-wide wake cost	not significant	1.16*10^-4 (progress continues)
		4.47*10^-4 (progress obstructed)

.

b – I(n,t), S(n) experimental values

64 megabyte - b	5 sec	10 sec
5026	+5*	+10*
5045	+5*	+10*
5045	---	---
5046	+1	+1
5047	+1	+1
5047	---	---
5067	+5*	+10*
5081	+5*	+10*
total sec	22	42
total spin*	4	4

64 megabyte - c	5 sec	10 sec
2928	+5*	+10*
2932	+4	+4
2965	+5*	+10*
2965	---	---
2971	+5*	+6
2977	+5*	+6
2977	---	---
2978	+1	+1
2984	+5*	+6
3019	+5*	+10*
3020	+1	+1
3024	+4	+4
3027	+3	+3
3027	---	---
3027	---	---
total sec	43	61
total spin*	6	3

96 megabyte - b	5sec	10sec
5026	+5*	+10*
5047	+5*	+10*
5047	---	---
total sec	10	20
total spin*	2	2

96 megabyte - c	5sec	10sec
2928	+5*	+10*
2932	+4	+4
2984	+5*	+10*
3019	+5*	+10*
3020	+1	+1
3024	+4	+4
3027	+3	+3
3027	---	---
3027	---	---
total sec	27	42
total spin*	3	3

9 – References

[Doug94] Fred Douglis and P. Krishnan and Brian Marsh. Thwarting the Power-Hungry Disk, In {USENIX} Winter, 292-306, 1994.

[Kue97] Geoffrey H. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. In Proceedings of the sixteenth ACM symposium on Operating systems principles, ACM Press, 264-275, 1997.

[Li01] Zhiyuan Li and Cheng Wang and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems, pages 238-246, ACM Press, 2001

[Rob90] John T. Robinson and Murthy V. Devarakonda. Data-cache management using frequency-based replacement. In Proceedings of the 998 ACM Sigmetrics conference on Measurement and Modeling of Computer Systems, ACM Press, 134-142, 1990.

[Ru98] A. Rudenko and P. Reiher and G. Popek and G. Kuenning. Saving Portable Computer Battery Power Through Remote Process Execution, In Mobile Computing and Communications Review, 2(1):19-26, 1998.

[Sat02] M. Satyanarayanan. The evolution of coda. In ACM Transactions on Computer Systems, 20(2):85-124, 2002.

[Tai95] Carl Tait and Hui Lei and Swarup Acharya and Henry Chang. Intelligent file hoarding for mobile computers, In Proceedings of the first annual international conference on Mobile computing and networking, ACM Press, 119-125, 1995.

[Yag00] Karim Yaghmour and Michel R. Dagenais. Measuring and Characterizing System Behaviour Using Kernel-Level Event Logging, Proceedings of the 2000 USENIX Annual Technical Conference, 2000.