General-Purpose Persistence Using Flash Memory

Jonathan T. Moore, Michael Hicks, and Scott Nettles

Computer and Information Science Department
University of Pennsylvania

April 21, 1997

Abstract

Flash memory is a solid-state semiconductor memory technology that has interesting price, performance, and semantic tradeoffs. We've developed Gordon, a general-purpose persistence system for Standard ML, that uses Flash mapped into the virtual address space as its stable storage medium.

Flash supports a write-once/bulk erase interface which makes it difficult to support update-in-place semantics. In addition, Flash chips are only guaranteed to survive a limited number of erase cycles. Gordon has been designed to overcome these difficulties, and our performance analysis demonstrates good performance and reasonable lifetimes for appropriate application domains.

1 Introduction

Flash RAM is a semiconductor memory which offers significant new price, performance, and semantic tradeoffs for the "main" memory of computer systems. Flash has read performance and density comparable to DRAM, but unlike DRAM, data stored in Flash is stable and is not lost on power failure. Compared to disk, the most widely used form of stable storage, Flash offers superior read performance, lower power consumption, better ruggedness, and shorter write latencies. Flash also has a key advantage over previous solid-state stable storage devices, such as CCDs or Bubble memory: Flash is already a significant part of the memory market. Flash sales are in the billions of dollars per year, and many major semiconductor manufacturers offer a Flash product. In short, Flash is real, has useful performance characteristics, and is a main-stream product that is here to stay.

Flash has some important disadvantages as well. Writing a single byte to Flash is two orders of magnitude slower than DRAM, although still three orders of magnitude faster than disk. More problematically, Flash cells are write-once until explicitly erased, and this erase may only be applied to a block of cells (64 kilobytes for the chips used here). To make matters worse, erases are extremely slow. This write-once, bulk-erase property means that Flash alone cannot support the standard update-in-place semantics of DRAM and disks. Finally, depending on the level of memory array parallelism, Flash may not provide the same write bandwidth as disk, especially if erases cannot be done concurrently.

There are a number of applications where Flash can be used in general purpose computers. It can be used as a replacement for (or extension of) main memory. For example, Wu and Zwaenepoel's system, eNVy [16], uses extensive amounts of Flash hardware to present Flash-based memory with in-place update semantics. Flash may also be exploited for its persistence and low-latency writes as a replacement for disk. Kawaguchi, Nishioka, and Motoda [4] have looked at implementing a Flash-based file system, and in a related work [8], we explored the use of Flash as a substitute for disk in a transaction log. Finally, a hybrid approach might be taken, using Flash to replace both disk and main memory in a single application. It is this approach that we explore here.

In this paper, we study the use of Flash that is directly integrated into the memory hierarchy and mapped into the user's virtual address space. This design allows the user to read Flash addresses directly. Further, because Flash is persistent, writes to Flash will survive a power failure. Of course, the user will be limited by Flash's write-once semantics and a method for erasing Flash blocks must be provided by the system.

We use this interface to Flash to implement a system providing general-purpose persistence. In our system, which we call Gordon, any data in a user's program can become persistent, regardless of type, simply by virtue of being reachable by pointer dereferencing from a special location called the *persistent root*. This makes it very easy for programmers to make data structures persistent, since any data structure can be made persistent without conversions from internal to external form. The entire system is garbage collected. We use a simple transaction system to allow the persistent data to be manipulated consistently.

We claim that our implementation motivates applications that may benefit from Flash's relatively fast read and write latencies. To show this, we compare the performance of Gordon to that of our previous system, Sidney, which uses disk to store the persistent data and either disk or Flash to store the transaction log. Our results show that indeed, Flash's fast read and write latencies allow Gordon to outperform Sidney. At the other extreme, we compare Gordon's performance to a commercial transaction system on a typical transaction load. As expected, we find that Gordon's performance is limited by bandwidth, and while respectable, it does not compare to high-end systems.

Our evaluation system contains no Flash memory, but because Flash has the read performance of DRAM, it is easy to emulate Flash accurately by using DRAM to store data and using timers to match write and erase performance to that of Flash. This arrangement allows us to explore a wider range of Flash design parameters than would Flash hardware and provides a conservative lower bound on the performance achievable by Flash. Our software guarantees that Flash's unique semantics are obeyed.

In the next section we motivate the types of applications we believe could take advantage of a Flash system like Gordon. In the two sections that follow we present the specific details of the Flash hardware we model, followed by its operating system interface. Then we describe the design and implementation of Gordon and how it uses Flash to support persistence. The next three sections detail the experimental setup and results of our benchmarks, and some future work. Finally, we wrap up with related work and conclusions.

Quality	DRAM	Flash	Disk
Read access (cell)	60 ns	70 ns	6.15 ms
Write access (cell)	60 ns	6 μs	6.15 ms
Erase time (block)	*n/a*	600 ms	*n/a*
Cost/MB	\$2.80	\$21.00	\$0.25

Table 1: Cost and Performance comparison of Flash, DRAM, and Disk

2 Motivation

The non-volatile nature of Flash means that it can be used as a lower cost and/or higher performance alternative to disk in systems that manipulate permanent storage. While its other characteristics may suggest other uses for Flash, we focus on its use as a replacement for disk.

Table 1 shows the performance and cost differences of Flash, DRAM, and disk[1]. We examine these contrasts in the following subsection.

2.1 Flash as a Disk Replacement

There are four possible reasons to make this substitution: reliability, power consumption, cost, and performance.

Because Flash is a solid state memory, it has no moving parts and is thus not susceptible to mechanical failure. This means that Flash may be used in some environments where disks cannot. Furthermore, disks generally fail catastrophically and unpredictably, whereas Flash fails by gradually wearing out. This allows its performance to be monitored and predicted, thus facilitating its replacement before failure occurs. Therefore, Flash is an attractive option for environments that can tolerate predictable failures but not catastrophic ones.

Operations to Flash consume significantly less power than disk. This is especially important for embedded applications and mobile computers. We don't present specific numbers here; the reader may refer to [10].

Flash can result in cost savings when only a small amount of permanent storage is needed. This is because the typical minimum storage size of a disk is much larger than the minimum of Flash, resulting in a lower total cost for Flash, despite its greater per-byte cost. This is why Flash is becoming popular as "upgradable ROM" in embedded applications such as modems.

[1] DRAM and disk data taken from typically available devices, Flash data from [2]

The most intriguing possibility, however, is using Flash to improve the performance of systems that support persistence. For reads, Flash latency is far superior to that of disk and has no bandwidth limitations, as shown in Table 1. Although the latency for Flash writes is high when compared to that of DRAM, it is quite low when compared to the latency of disk writes. For relatively small writes and infrequent erases, this means Flash has a significant speed advantage over disk.

However, using Flash rather than disk may result in a significantly lower overall write bandwidth, due to the need to periodically perform lengthy erases. This means that for writes that are sufficiently large or frequent, Flash will be outperformed by disk despite the write latency advantage. Of course, this problem could be overcome by using more Flash, which would make erases less frequent. However, the lower cost per-byte of disk makes this option unattractive.

It is important to understand that this implies that Flash does not offer a fundamental advantage in typical high-performance commercial transaction systems. In those systems, there is a high degree of concurrency which allows several transactions to be queued up and committed all at once (a technique known as *group-commit*). This allows a disk's bandwidth to make up for its high latency, and then its cost and update-in-place semantics make it more attractive than Flash. Unfortunately, this technique of batching commits is not useful in applications with little or no concurrency, or whose writes are infrequent.

As an alternative to Flash, Gordon could easily employ battery-backed DRAM as its persistent medium. Papers that have addressed this possibility in the past have usually dismissed it due to high cost. However, with current memory market DRAM prices as low as \$4 per meg, and battery costs of \$10-\$15 for reasonably long-lived lithium batteries, this option may be attractive for some applications. Clearly for applications requiring extreme reliability, Flash is better suited than is DRAM, since DRAM batteries may expire. DRAM-based systems also tend to be bulkier, may employ a disk, consume more overall power, and are only available in custom setups. For these reasons, we believe that Flash's simplicity and its current market status will make it more often applicable.

2.2 Applications

Based on the above observations, there are two broad categories of applications that seem like good customers for Flash. The first are applications where the ruggedness and cost saving possibilities are significant—typically, embedded systems. The other will be systems in which there is a significant advantage to having fast Flash writes, i.e. systems that need high performance updates of permanent data, but which do not have significant concurrency to allow batching. We consider an example of an application from each category below. Each of these applications will be used as benchmarks to test our implementation in Section 8.

Reliable Communications Components

Our first example is motivated by a commercially deployed, real-world application with which one of the authors has direct experience. In this system, the Aircraft Communications Addressing and Reporting System (ACARS) [1], an airplane broadcasts important telemetry and accounting data over a radio frequency. This information is picked up by a nearby *groundstation* which relays the message back to a central processor over a ground-based network. The central processor then returns an acknowledgement which is transmitted by the groundstation. Because groundstations may live in harsh and remote environments, such as atop mountains, the use of the rugged hardware is a requirement. Therefore, no disk is used in the current arrangement. This means that a groundstation failure may result in lost messages. Losing messages will cause the service provider to pay significant penalties, and may have safety-related consequences as well. One way to prevent this is to save each message to permanent storage before it is relayed to the central processor. After the groundstation has transmitted the acknowledgement, it may free the storage. Flash is ideally suited to this purpose. It is rugged and will not fail unpredictably. Only a small amount of Flash is needed: just enough to store the maximum number of outstanding messages (plus some more to help amortize the erase times). Since there is essentially no concurrency in the system, the rate at which the messages can be committed is directly related to the write latency of the persistence system.

Persistent Productivity Tools

Our second example is taken from the class of productivity tools that see heavy use on personal computers, such as text editors, drawing programs, spreadsheets. Currently, users protect themselves from machine failures by saving frequently to disk. Unfortunately, saving to disk often takes several seconds and disrupts the flow of work. An ideal solution would be an application which would allow recovery from a crash to the exact state before failure. This would be possible if the program's entire state could be made persistent and committed after each keystroke or mouse click. The underlying persistence system would clearly need minimal write latency to be unintrusive. This requirement exactly matches the semantics of Flash. Considering the frequency at which current personal computers crash, this exact recovery would be a wonderful marketing feature.

3 Flash Memory Hardware

Flash is a variant of EEPROM where individual-cell erases have been replaced with block erases. This allows manufacturers to design Flash cells using only one transistor, thus achieving DRAM-like densities. Internally, Flash is organized similarly to DRAM and Flash chips are available that have standard DRAM interfaces [7], making it straightforward to use Flash in memory subsystems originally based on DRAM.

The Flash chip we consider in this paper is the 16-megabit Intel 28F016SV [7]; its performance characteristics are presented in table 1 next to those of typical DRAM and disk. These chips are among the largest Intel offers. They are manufactured with a similar technology (0.6 micron process) and at the same density as standard DRAM. The chips can be configured as 2M by 8 bits or 1M by 16-bits; in our system, we use the former arrangement. Each chip has 32 independently erasable blocks of 64 kilobytes each. There is no on-chip parallelism; this is provided by using bank of Flash chips, as described in the next section. There are a few other details about the 28F016SV that we leave to Section 10, Future Work, as they are not pertinent to our current system.

In general, Flash chips can be read at standard DRAM speeds, while writing them is several orders of magnitude slower. Furthermore, once a bit in Flash has been changed from its initial value of one to zero, it can not be changed back without erasing a large block of memory. However, it still remains possible to change ones to zeroes, since writing Flash ANDs the current contents of the memory with the new value. Although this ANDing gives somewhat more flexibility than if Flash were merely write-once, we do not take advantage of it in Gordon.

When Flash is erased, a large block of cells is set to one. This is an extremely slow operation: on the 28F016SV, erasing one 64KB block requires 600 milliseconds. This limited erase bandwidth may affect overall performance, as we discovered in our previous work using Flash for transaction logs. It is therefore important to find ways either to reduce an application's erase bandwidth requirements or to overlap Flash erases with other operations. The chip does facilitate an operation called *suspend* which can suspend an outstanding write or erase to one cell or block, read a cell from a different block, and then resume. How this operation might be used is explored below.

Writing and erasing Flash cells "stresses" the semiconductor oxide, increasing the time for writing and erasing. Eventually, the time to write or erase becomes long enough that the memory is considered to have "failed", although this does not result in the loss of any data. Thus the observed failure mode of Flash is predictable and not catastrophic, unlike disk failures.

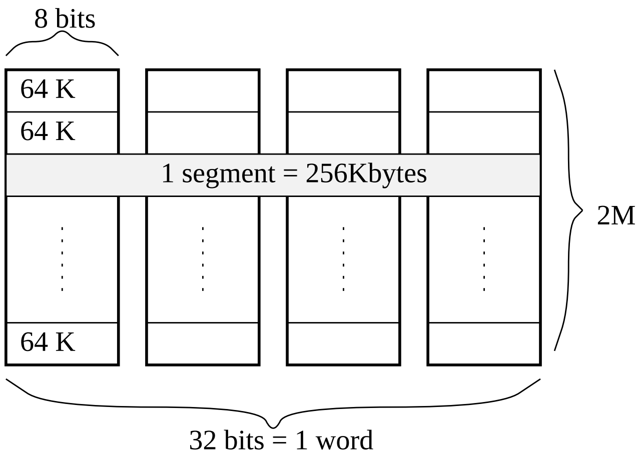

Figure 1: A Flash Memory Bank with 4-way Parallelism

How long a Flash chip will last before it "fails" is directly correlated with the erase blocksize: the larger the blocksize, the longer the lifetime. Current Intel chips, among the most reliable on the market, are guaranteed to survive 1 million erase cycles per block [3], although laboratory experiments have shown that Flash can often outlast its guaranteed lifetime through the use of wear-leveling and block retirement, as discussed in Section 8.2.

3.1 Flash Parallelism

Flash chips may be arranged in parallel to form a bank, similar to modern DRAM. Memory addresses are "striped" across several Flash chips, essentially increasing write and erase bandwidth by the level of parallelism. Figure 1 illustrates a parallel bank of four 28F016SV chips that presents a 32-bit wide interface to 8 megabytes of Flash; write and erase bandwidths are increased four-fold from single chip levels. In fact, much greater levels of parallelism are not unreasonable; the eNVy system [16] implements 256-way parallel Flash banks.

This performance improvement comes at a higher total cost and reduced flexibility. Because data is striped across the chips, the effective minimum eraseable amount of Flash is increased. This defines what we shall call a Flash *segment*. In the 4-way case, these segments are 256 kilobytes, as illustrated in the Figure. In the 256-way parallel case, the segment size would be 8 megabytes, with an overall bank size of 512 megabytes. At current prices, such a bank would cost $10,700. Thus, for applications that require only a small amount of Flash, such as a reliable communication component as described in Section 2.2, only limited parallelism may be used if cost is to be kept to a minimum.

Operation	Inputs	Result
`Flash_alloc`	*num, addr*	maps Flash buffer of size *segmentsizeB* × *num* at address *addr* and returns a *persistent key*
`Flash_free`	*addr*	disassociates the buffer allocated at *addr* from the application's *persistent key*
`Flash_erase`	*addr, num*	erases *num* segments of the Flash buffer starting at *addr*
`Flash_recover`	*persistent key*	maps Flash associated with *persistent key* into the address space

Table 2: An operating system interface for Flash

4 Flash OS Interface

This section describes one operating system interface to Flash memory that allows programs to map Flash into their virtual address space, where it may be directly addressed. We have not actually modified an operating system to provide this interface nor do we use actual Flash memory; rather we have written Gordon to simulate this interface and enforce Flash's semantics. We wanted to demonstrate that real applications could perform well using this interface before actually building it. This arrangement allows us to model a range of interface performance characterstics, rather than ones specific to a particular Flash chip or VM implementation. We describe the details of this simulation here.

4.1 OS Interface

We assume that the OS manages a physical pool of Flash memory and provides the user access to this Flash in the user's virtual address space. The interface consists of four operations listed in Table 2.

We chose this interface because it seems to be the simplest interface that provides the system services we require. `Flash_alloc` maps *num* segments of already-erased Flash starting at the virtual address *addr* and returns the application's *persistent key*. The parallelism of the Flash bank will create a lower bound for the segment size, as described in Section 3.1. `Flash_recover` restores the Flash associated with the *persistent key* to its old location(s) in the virtual address space. For simplicity, we can assume each application will always have at most one persistent key assigned to it, and all blocks of allocated Flash are associated with that key. `Flash_free` is used to disassociate a block of Flash from the application's persistent key. Finally, `Flash_erase`, erases *num* segments of Flash beginning at address *addr*.

This interface allows the operating system to manage Flash much like DRAM-backed virtual memory but on the granularity of Flash segments. If needed, the system can page Flash out to disk. In addition, the OS is be responsible for detecting the failure of Flash blocks and then retiring them from the free pool. This is easily accomplished by comparing the time to erase a block with some pre-defined threshhold. Finally, the OS must (persistently) keep track of the map of persistent key to physical Flash segment(s), so that data may be properly recovered following a crash.

To support the semantics of `Flash_alloc`, the OS must work to keep its free pool of Flash in an erased state. Thus, to optimize the time `Flash_erase`, it may simply map out the dirty Flash to be erased in the background, and then map in some already-erased Flash from its free pool. Of course, if there is not enough erased Flash available, the call will block until there is. To model these situations, our performance analysis considers the two extremes: the case when there is always enough erased Flash, so that `Flash_erase` costs only the time to remap, and the case when there is no pre-erased flash, in which case the cost to the user is the physical erase time plus a small overhead.

4.2 Flash Emulation

We use DRAM as the basic storage of our Flash emulation. Since Flash has the same read performance as DRAM, when reads from Flash are needed, we simply read the DRAM. When a write or erase is needed, we set a timer for the duration of the operation, perform the operation in RAM, and then wait for the timer to expire, at which time the system resumes execution. We have been careful to ensure that Gordon preserves Flash's write-once semantics.

There are other details involved with actually implementing Flash in hardware that we have not covered here since they don't pertain to the accuracy of our emulation; we address them in Section 10.

5 System Overview

Gordon implements a general-purpose persistence interface for the programming language Standard ML (SML), using the Flash OS interface described in the previous section. Gordon is implemented in much the same way and exports the same persistence interface as its predecessor system, Sidney.

The persistence interface provides four operations to the user: `alloc` allocates (and initializes) a new value in memory; `read` reads a value found in memory; `write` modifies, or *mutates*, a value found in memory; and `commit` forces all changes to persistent data to become stable. There are also operations that allow the programmer to make specific data reachable from the *persistent root*, which will be described below.

In this section, we discuss how Flash memory relates to the major features of the system, in particular, Flash's relationship to general-purpose persistence, certain characteristics of SML, transactions, and generational copying garbage collection.

5.1 General-Purpose Persistence

A system that provides general-purpose persistence essentially allows any kind of data to be made permanent. This is in contrast to the special-purpose persistence that is provided by filesystems and databases, in which only certain types of data (respectively, byte streams and database records) can be made persistent.

Gordon implements two related forms of general-purpose persistence—*orthogonal persistence* and *persistence by reachability*. The property of orthogonal persistence means that whether or not data is persistent is independent of (or is orthogonal to) the type of the data. That is, in an orthogonally persistent system there is no representational difference between persistent and non-persistent instances of a given data type. The complementary notion of persistence by reachability defines persistent data to be anything accessible from pointer dereferences starting at a special location, called the *persistent root*. This means that in Gordon, making a datum persistent is as simple as installing a pointer to it in some other already-persistent data structure.

5.2 Standard ML

Gordon implements a persistence interface for Standard ML (SML). One important feature of the SML implementation upon which Gordon is based (SML/NJ version 0.75) is that heap objects have header words attached to them which makes it possible to distinguish mutable objects from immutable ones. Since SML is functional language, we would expect that most of a typical program's data is immutable.

These characteristics make Flash very suitable for implementing persistence for SML. Immutable persistent data can be addressed directly in Flash; the fact that no updates will occur to this data fits nicely with Flash's write once erase semantics. Mutable persistent data, although uncommon, is still problematical, since we cannot overwrite a Flash cell without first erasing many neighboring cells. We address this problem in the following section.

5.3 Transactions

Providing support for transactions is a critical part of supporting persistence. The reason is simple: if the programmer cannot control when changes to data become permanent, then it becomes very difficult to make sure that the data is in a consistent state upon system failure and recovery. Transactions provide an operation called *commit* which allows a consistent set of updates to become the recoverable state atomically. Some systems provide the ability to *roll-back* or *abort* a transaction, or may allow *concurrent transactions*; although our system does provide the basic features that are necessary to support these operations, we do not discuss them here.

The standard method [6] for implementing transactions is not to write updates to the actual persistent data, but rather to write updates to an append-only log stored on disk. These writes are easy to make atomic and typically do not involve seeks (or only seeks to a nearby track) thus incurring only a rotational delay. The transaction log can then be "replayed" on recovery to construct a consistent image of the persistent data. When the log gets too long, it is compacted (or *truncated*), and the old log is thrown away.

In our previous work, we studied the use of Flash as an alternative to disk for transaction logs. Flash is ideally suited for this, since its write once/bulk erase interface maps directly to the append-only/truncate operations of the log. Furthermore, Flash's lower write latency, when compared to that of disk, can offer a performance advantage. Gordon also uses a Flash-based transaction log to store updates to mutable persistent data. More detail is provided in the next section.

It is important to understand some of the areas of transaction processing with which Gordon is not concerned. In particular, it is not intended for use in high-performance transaction systems, such as those used by banks and airline reservation systems. These systems are highly concurrent and this concurrency allows them to take advantage of group-commit to improve throughput. Gordon, on the other hand, is intended for environments in which disks are not usable, in which write latency is critical, or in which there is insufficient concurrency for batching commits.

5.4 Generational Copying Garbage Collection

Gordon manages a program's data using copying garbage collection (GC). GC is an automatic storage management technique which locates pieces of dynamically-allocated memory that are *garbage* (no longer accessible by the program) and reclaims them. The collector starts with a set of *roots*, typically pointers in registers or in the stack. By tracing the pointer graph from the root set, it is possible to locate all of the memory which the client program might access (the *live* data); anything else is garbage and can be reclaimed. The time for required for a copying GC is therefore correlated with the amount of live data, referred to as the *livesize*.

In copying garbage collection, a program's heap is divided into two areas, only one of which is active at a time. Allocation occurs in a linear fashion until the current space fills up. At that point, the collector locates the live objects and copies them to the other space, compacting them as it goes. By adjusting the pointer graph and the root set to be consistent with the new locations of the objects, the garbage collector can then throw away the entire first space. The client program then continues, allocating to the top of the second space until another GC is needed. One typical side effect of copying GC is that as each object is moved, a *forwarding pointer* is left in its old location.

Gordon's GC also employs *generational collection*. A generational collector segregates objects by allocation age into heaps (called *generations* which are collected separately. As younger objects survive collections, they are *promoted* to older, less-frequently collected generations. This exploits the observed property that younger objects are more likely to become garbage, thus reducing the number of times an object is copied between spaces.

Again, the write-once/bulk erase semantics of Flash make it suitable for management by copying GC. In particular, when we reclaim one of the two spaces from a collection, we can erase it entirely. At the next collection, we can then copy live objects back into the Flash without violating the write-once semantics. In addition, because allocation occurs linearly, we will have written to almost every cell of the Flash which we erase. As we will discuss in Section 8.2, this is important to efficient use of Flash memory.

For more information on copying and generational collection, see the survey by Wilson [15].

6 Implementation Details

In this section we describe how we implement Gordon's persistent interface. First, we present the details underlying heap structure used by the application and how it is garbage collected. We then describe how `commit` is implemented. We wrap up with a few more details about the process.

6.1 Application Heap Layout

Gordon implements the heap used by the application program by dividing process heap's virtual memory space into two logical sub-heaps: the *transitory heap* and the *persistent heap*. The data in each of these heaps is directly accessible by the application program. The arrangement of these heaps is illustrated in Figure 2, and will be described in detail below.

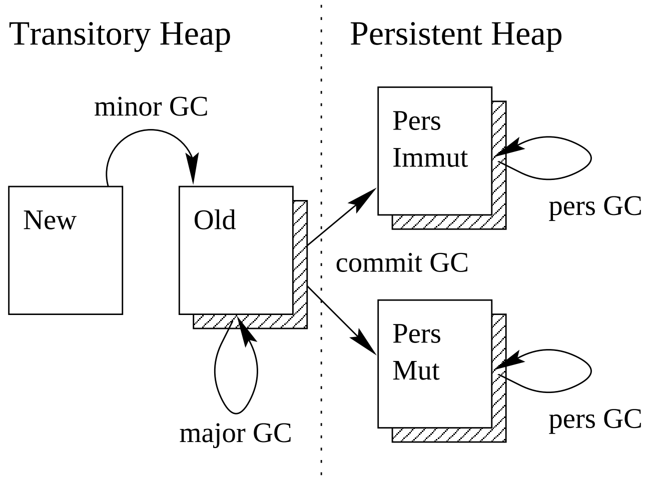

Figure 2: The Heaps of Gordon

The Transitory Heap

The transitory heap consists of all data that need *not* be recovered after a failure, and thus is stored in DRAM. It is composed of two generations, called *new space* and *old space*. All allocation for the application occurs in new space. When the new space reaches a certain capacity, as determined by the limit pointer, it is garbage collected into old space; that is, all reachable new space objects are promoted into old space and the allocation pointer is reset. This is called a *minor GC*. Following this collection, the capacity of old space is checked. If need be, it is collected as well, termed a *major GC*.

The Persistent Heap

The persistent heap consists of all data that has (at some point) been committed. Since this data cannot be lost on failure, it must have some representation in Flash. This is where we take advantage of SML's distinction between mutable and immutable data. Since immutable data will not change after it has been initialized, committed immutable data may be stored into Flash and then directly accessed by the program; the Flash used for this purpose is called the *persistent immutable space*. On the other hand, application `writes` may result in updates to the persistent mutable data, so it must be stored in DRAM; this space is called the *persistent mutable space*. A persistent representation of this data is stored in a Flash transaction log called the *persistent log*, which is described in detail below. Like the transitory spaces, the persistent spaces may fill up and require a garbage collection, called a *persistent*

GC. Garbage collecting the Flash-based persistent immutable space presents some interesting problems, as we shall discuss in Section 6.3.

6.2 Commit

`Commit` must do three things. One, it must find any data that is located in the transitory heap, but which is reachable from the persistent root, and move it into the persistent heap. Second, it must stablize any changes made to the persistent heap. Finally, it must guarantee that all changes to the stable store are atomic, so that the application data may be recovered in a consistent state.

Commit GC

To find the newly persistent data that is located in the transitory heap, we make the conservative assumption that all data in the persistent heap is reachable from the persistent root. This implies that any pointers from the persistent heap to the transitory heap point to data that should be made persistent. Furthermore, any data pointed to by those data should also be persistent. Now the problem is exactly the one solved by copying garbage collection: start with a set of root pointers, find all reachable data, and copy it elsewhere. We call the GC that moves newly persistent data from the transitory heap to the persistent heap the *commit GC*. Notice that this collection must copy mutable data to the persistent mutable space in DRAM and immutable data to the persistent immutable space in Flash, as illustrated in Figure 2. Therefore, while the immutable data is now persistent, the newly copied mutable data is not; it must be recorded in the persistent log.

Commit GC effectively *promotes* transitory objects into the persistent heap, as in generational collection. As a result, some pointers in the transitory heap may become *stale*: they point to the old locations of objects that have been moved into persistent space. Therefore, after the commit GC, Gordon must fix these pointers. This can be acheived as a side effect of a major GC immediately following a commit GC.

The Persistent Log

Because the persistent immutable space is already stable, the persistent log need only track updates to the persistent mutable space. These updates will either be mutations to previously committed data, or data newly acquired by commit GC. The log consists of two spaces, so that when the first space fills, the second space may be used without incurring a synchronous erase. Each entry consists of a location and length pair followed by the data to be restored to that location upon recovery.

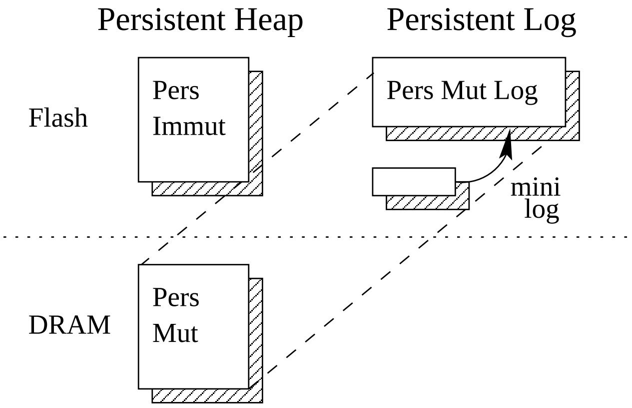

Figure 3: The Persistent Space

Atomicity

To make `commit` atomic, we only need to guarantee that data is stored to the persistent log atomically. This is because any data copied to immutable persistent space by commit GC could only have been made reachable by an update to the mutable persistent space, which will be tracked by the log. Updates to the persistent log are made atomic by storing the end-pointer of the persistent log into another log, called the *mini-log*, also in Flash, as the final act of `commit`. On recovery, the persistent log is replayed only up to this position, guaranteeing the atomicity of each transaction. The mini-log itself consists of two spaces for the same reason the persistent log does, and Gordon guarantees that the transition between these spaces is atomic. Note that the size of a mini-log space is lower-bounded by the Flash segment size.

All of the data structures we've discussed to support persistence in Gordon are referred to as the *persistent space*. This is illustrated in Figure 3.

6.3 Remaining Issues

We have glossed over three issues which we shall now briefly touch upon. One is how we determine which heap locations have been updated between commits and garbage collections. Another is how we determine when a space should be collected. The last is how we garbage collect Flash.

The value and location of each `write` made by the application is recorded inline to a *store-list* maintained by the compiler. This store-list is valuable for a number of reasons. One, it faciliates a speedy determination of nonstandard roots for both major and minor GC. Two, it provides `commit` with the information it needs to log updates to the persistent heap. Finally, it can be used to support abort by also tracking the old value of the write.

The garbage collection of generation is determined by a *limit pointer*; therefore there exist new space, old space, and persistent heap limit pointers. When a space's data exceeds or is about to exceed its limit pointer, it must be collected. How this limit pointer is set is a matter of policy. In Gordon, the limit pointers are provided as runtime parameters in terms of the number of new bytes allowed before initiating a collection.

There are three primary issues with the garbage collection of Flash. One is that writing a copied object's forwarding pointer in-place is prohibited by the write-once semantics of Flash. To get around this, we instead hash these forwarding pointers into a large table stored in DRAM. The second problem is how to update pointers that reside in Flash. The key observation here is that such updates only occur in the space being copied to. Therefore, when the destination space consists of Flash, we can GC into a DRAM scratch space instead, moving the final result en masse into the Flash space. There are other possible solutions, but these are the most straightforward, and not costly, as we shall see. Finally, it is important to point out that following a persistent GC, the original space must be erased to prepare for the next GC. We postulate that if this erase cannot be done in the background, it will dominate persistent GC time.

7 Experimental Setup

In order to examine the performance of Gordon, we designed a series of benchmarks intended to stress our system in various ways. The first two benchmarks are selected from the expected application domains we have described in Section 2.2. The third was designed to favor a disk-logging system so that we could study Gordon's applicability outside the above domains.

We measured our benchmarks over a variety of modelled Flash parallelism, from the 4-way case for our embedded system up to the 32-way case we might expect to see in a general computer. This results in minimum Flash size requirements ranging from 8 MB to 64 MB, respectively, which is reasonable given current Flash pricing. The relevant performance characteristics for each level of parallelism are summarized in Table 3.

To measure the peak performance our benchmarks might reasonably achieve, we also modelled a version of Gordon to use battery-backed DRAM. This version takes advantage of faster read and write times, no erase delays, and update in-place semantics.

In the remainder of this section, we discuss the conditions under which our experiments were conducted. For two of the benchmarks, we measured not only Gordon's performance, but also that of its predecessor, Sidney [12]. Therefore we begin by briefly describing the pertinent details before proceeding on to a description of the benchmarks and our experimental environment.

7.1 Sidney

Sidney presents the same programmer interface as Gordon, and differs only in the implementation of the persistent space: the persistent heap resides entirely in DRAM and is supported by a low-level transaction system called Recoverable Virtual Memory (RVM) [11]. RVM allows disk segments to be mapped into virtual memory and allows changes to these VM images to be transferred to disk atomically. Efficiency and atomicity are provided by the use of a disk-based circular log. When the log is full, it is truncated by applying the updates to the actual disk segments. In our experiments, we use a dedicated raw disk for the RVM log.

Some of our previous work involved modifying RVM to use a Flash-based transaction log rather than a disk-based one. We found that Sidney had substantially improved performance when using the Flash log. Therefore, we primarily compare Gordon to Sidney with the modified RVM, although the performance numbers for Sidney with the disk-based log are also provided for reference.

7.2 Benchmarks

Our three benchmark programs include a model of a persistent ACARS [1] groundstation, a crash-resistent text editor, and a transaction processing load based closely upon TPC-B [14]. The following subsections outline the relevant features of these programs and a standard benchmark we use to test them. Details about the Flash parallelism modeled are also provided.

7.2.1 Persistent Groundstation

Recall from section 2.2 that an ACARS [1] groundstation is responsible for reliably forwarding messages from airplanes to the central host. In order to guarantee that received packets are not lost, we `commit` each packet to stable storage before attempting to forward it. Since the central host generates the acknowledgment and not the groundstation itself, it is important that this process have the shortest latency possible.

Unlike typical SML programs, the groundstation program is written in a mutation-intensive style, keeping pending packets in a persistent queue. Indeed, during a benchmark run of 10,000 packets sized at 1500 bytes each, each `commit` copies 1576 bytes of mutable data but only 76 bytes of immutable data. We simulate infinitely fast networks on either side of the groundstation to test its maximum throughput.

Since the groundstation is an embedded application, we would like to use the least amount of Flash possible. We model 4-way parallelism, which would require only 8 MB of our Intel chips, and would allow us to present a one word wide interface. For comparative purposes,

Parallelism	writes latency	writes bandwidth (KB/s)	segment size	erases latency	erases bandwidth (KB/s)
single	6 μs	163	64 KB	600 ms	107
4-way		651	256 KB		427
8-way		1302	512 KB		853
16-way		2604	1 MB		1707
32-way		5208	2 MB		3413

Table 3: Write and Erase performance for various Flash parallelisms

we also study the effect of larger total Flash sizes on performance and lifetime.

The standard benchmark models the receipt and forwarding of 100,000 packets of 1000 bytes each with an average persistent queue size of 5.

7.2.2 Crash-Resistant Editor

Our second benchmark program is a simple curses-driven text editor which supports an unlimited undo capability. After each user keystroke, the entire state of the editor is committed by setting the persistent root to point to a record containing the current edit buffer and the current list of undo information.

The standard benchmark uses two copies of the IRIX 6.2 `mmap` manual page as the source of input keystrokes. Due to the functional style in which the editor is written, this benchmark `commits` only a small amount of immutable data (approximately 376 bytes) per key. This is possible because the compiler allows sharing of immutable data, which means that most of the current buffer is already in persistent space and need not be copied by the commit GC.

We study higher Flash parallelism here than in the groundstation benchmark; numbers are presented for 8-way, 16-way, and 32-way parallelism. Although current computers can support much higher overall memory parallelism, the minimum 64 MB of Flash for the 32-way case already costs over $1300 at today's Flash prices. However, in Section 10, we discuss the possibility that Flash prices will drop in the future, so our tests still provide useful performance insight.

7.2.3 Debit-Credit

Our final benchmark program adheres quite closely to TPC-B [14], with the exception that it does not scale the size of the database to match the transaction processing speed achieved by the system. This program, which we call Gordon Debit-Credit, is essentially an SML version of a C++ benchmark used to evaluate RVM [11].

For each transaction, we randomly select a target account. The per-account workload involves modifying three 128-byte database records and creating a 64-byte audit trail record. The standard benchmark runs Debit-Credit for 100,000 transactions in an attempt to evaluate Gordon's performance on an actual transaction-processing workload.

As in our editor benchmark, we measure 8-way, 16-way, and 32-way Flash parallelism. However, numbers are presented only for the 16-way case because we found that varying the parallelism did not significantly impact Gordon's relative performance.

7.3 Benchmarking Environment

For our benchmarks, we used an SGI Challenge-L. The machine is equipped with four 250 MHz R4400 processors and has 384 megabytes of main memory, which is two-way interleaved. Each processor has a first level cache that is split between instructions and data, with each cache being 16 kilobytes and direct mapped, as well as a second level cache that is unified, 4 megabytes in size, and also direct mapped. The machine runs IRIX 6.2.

The disks used rotate at 7200 RPM, or 120 rotations per second, and are connected to the machine using fast and wide SCSI. The machine can sustain a transfer rate of about 6 megabytes per second to a raw disk, and can write to the file-system at a rate of about 2 megabytes per second. For comparison, 4-way parallel Flash has a write bandwidth of .70 MB/s and 32-way Flash has a bandwidth of 5.6 MB/s when erases are ignored.

For Sidney, we stored RVM's data segment on a IRIX EFS file-system. The disk storing the data segment was only used by the benchmark. When testing the disk-based log, we stored the log on a raw disk, which was also dedicated to benchmarking. We found this configuration to provide the best performance.

We ran each of our measurements 11 times on a lightly loaded machine and we report the median of these values. All measurements are based on elapsed time. We observed no evidence that other processes disturbed our benchmarking; this is not surprising, given the single-threaded nature of our tests and the machine's overall capacity. For the final paper, we plan to isolate the machine from the network and idle all non-benchmarking

Flash Parallelism	System	Average commit latency	keystrokes/ sec
8-way	Gordon	2.0 ms	408
	Sidney Flash	2.5 ms	319
	Sidney Disk	47 ms	20
16-way	Gordon	0.9 ms	734
	Sidney Flash	1.6 ms	513
	Sidney Disk	45 ms	23
32-way	Gordon	0.6 ms	1068
	Sidney Flash	1.3 ms	593
	Sidney Disk	36 ms	28

Table 4: Comparison of `commit` latency and keystroke bandwidth for Gordon and Sidney

activity, although this seems unlikely to significantly affect our results.

8 Results

Our experimental results support the major claims we have made regarding Gordon's performance. In particular, `commit` times in our system have very low latency (particularly when compared to disk access latencies). At low levels of Flash parallelism, we can achieve acceptable throughput and long system lifetime for certain embedded systems applications. In addition, at sufficient levels of parallelism, Gordon can achieve significant throughput, although the necessary Flash configurations are not cost-effective. Nevertheless, we demonstrate Gordon's clear usefulness in certain application domains.

We found that each of the topics of latency, Flash lifetime, and bandwidth considerations were illustrated quite well by one of our benchmark experiments. We therefore present the performance of each benchmark followed by its corresponding topic below.

8.1 Crash-Resistant Editor

Because personal productivity software is often interactive, it make certain demands on a system's latency. Our crash-resistant editor was designed to stress Gordon in this way. In particular, the editor essentially makes its entire program state persistent after every input keystroke. This forces `commit`s to have low latency so that the user can make reasonable progress.

`Commit` Latency

We claimed in section 2.2 that one primary motivation for the use of Flash is its low write latency compared to that of disk. This motivation was present for both the persistent groundstation and the crash resistant editor. To confirm this assertion, we compared the average `commit` latency of Gordon under several Flash configurations with that of both the Flash and the disk versions of Sidney.

Table 4 summarizes our findings for this benchmark. Note that since the limit pointers were dictated for the Flash systems, we ran the Sidney disk system using the same limit pointers in order to facilitate comparison.

Considering that humans typically cannot detect pauses of less than 50 ms, we can see that performing a `commit` on every keystroke is not disruptive, even for the disk-based Sidney. Furthermore, the bandwidth requirements of the application are also easily satisfied; typing rates of 1068 keys/sec (which translates to more than 10,000 words per minute!) are supported. This throughput suggests that more complicated persistent productivity applications, such as CAD tools or digital image manipulation might be reasonably supported – particularly because Gordon is well under the 50 ms threshold. The same could not necessarily be said of the disk-based Sidney.

Notice that the Flash-based Sidney also has quite good latency times, due to the fact that `commit` can complete as soon as the changes have been permanently logged. However, this version of Sidney is still backed by the disk-intensive RVM, and therefore achieves lesser overall throughput.

While Gordon's average keystroke response time is fast, the keystroke that initiates a persistent GC is extremely slow; the elapsed time of the average persistent GC is 4 seconds. However, the key observation here is that persistent GC time is dominated by the time to write and erase Flash. Running Gordon with simulated zero-latency Flash writes and erases provides a persistent GC elapsed time of 1 second, but this is still disruptive (although certainly more bearable).

This suggests that Gordon would significantly benefit from concurrency. At the very least, if Flash erases could be done in the background by the OS as described in Section 4.1, and the persistent GC were made concurrent, we would hope to nearly eliminate the disruptive pauses. We discuss this possibility in our plans for future work (Section 10).

It is worth mentioning here that based on information we shall present in the next section, the expected lifetime of the crash resident editor at full bandwidth for the 8-way parallel case (assuming background Flash erases) would be 387 days. Considering that very few humans indeed can type at 408 keys per second (even 20 keys per second is quite fast), it is likely that the resulting 20 year expected lifetime (assuming a user could type continuously for that period) is more than sufficient.

8.2 Persistent Groundstation

One important characteristic of the groundstation benchmark is that the queue of pending packets never exceeds a small, maximum threshold. This means that the amount of live data in the heap remains stable over time. Furthermore, since the packets arrive at a constant rate, the persistent heap likewise fills up at a constant rate. These facts will be important to our understanding of Gordon's system lifetime for this benchmark.

Flash Lifetime

The lifetime of Flash in a system is essentially n/r, where n is the number of erase cycles that the Flash configuration can endure, and r is the rate at which an application requests that bytes be erased, which we term the *demand erase bandwidth*. System lifetime can thus be lengthened either by increasing n or by decreasing r.

Intel guarantees that the 28F016SV will read and erase at or under specified times times for 1 million erase cycles per block. Since we have 32 blocks per chip, this works out to 32 million erases per Flash chip. In a tech report [9], Intel further asserts that this figure is conservative since it has to assume heavy use of single blocks in order to meet the guarantee. More evenly-distributed erasing of blocks, or *wear-leveling*, would essentially allow even higher erase cycle lifetime It is easy to imagine that the OS interface presented in Section 4.1 could maintain a small amount of data (also in Flash) to guarantee nearly perfect wear-leveling of the Flash pool: requests for `Flash_alloc` or `Flash_erase` would be satisfied by the least-used Flash pages first. Furthermore, since Gordon has linear growth of its persistent spaces, our persistent data effectively marches across Flash, potentially achieving excellent wear-leveling even without OS assistance. However, to be conservative, we make our lifetime calculations assuming a value of 32 million erases per chip.

The rate at which an application will require erases is proportional to the frequency with which the persistent heap fills up, since this is when persistent GC's (and hence erases) are triggered. How quickly a fixed-size heap will fill up is determined by the initial amount of data which survived the last collection, the rate at which new persistent data is moved into the heap, and a factor we call the *erase efficiency*, which is simply the percentage of bytes in a Flash segment which were actually written before being erased.

Because data is added to the persistent heap in a linear fashion, all of the segments in the heap, except possibly the last, will have perfect erase efficiency. Therefore, by setting the limit pointer to fall on a segment boundary, we would expect to achieve the highest efficiency for the last block. Note that the smaller the number of segments in the heap, the more the last segment's imperfect efficiency will affect that of the entire heap, as observed by varying the limit pointer from the maximum heap size, the effects of which are summarized in Table 5[2].

Heap Size (segments)	Maximum Heap (segments)	Erase Efficiency
1	4	53%
2	4	65%
3	4	73%
4	4	88%

Table 5: Flash erase efficiencies in terms of segments per heap

Parallelism	Demand Erase b/w (KB/s)	Demand b/w (free erase) (KB/s)	Max Array b/w (KB/s)	Lifetime (days)
4-way	210.3	414.6	426.7	228
8-way	357.7	584.2	853.3	324
16-way	540.8	789.6	1706.6	480

Table 6: Parallelism and Heap Size and their effect on Demand Erase Bandwidth

Therefore, doubling the heap size should reduce the persistent collection frequency by approximately the same factor. However, we would also be forced to erase twice as much Flash, so we would expect the erase demand bandwidths in Table 6 to be the same across heap sizes. The explanation for the differences is that by doubling the amount of Flash, we also double our effective write bandwidth, which means that the heaps fill up twice as fast. The fact that the demand bandwidths in Table 6 do not increase linearly with parallelism is simply due to the fact that the groundstation does not spend all of its time writing to Flash.

Table 6 also shows the measured erase demand bandwidth measured if erases could always be performed in the background by the OS. Notice, in fact, that these rates could be supported by the maximum erase bandwidth capabilities of the measured parallelism. The lifetime calculations in this table are based upon the these higher "erase for free" bandwidth demands.

These lifetime statistics seem rather disappointing, at first, although they are still conservative in two ways. Secondly, even the 4-way parallel case supports a throughput of 159 packets per second, at a packet size of

[2] Unfortunately, Gordon inherited a policy from Sidney which did not cause the limit pointer to fall on Flash segment boundaries. Due to time constraints, we were not able to rerun our experiments with an improved policy.

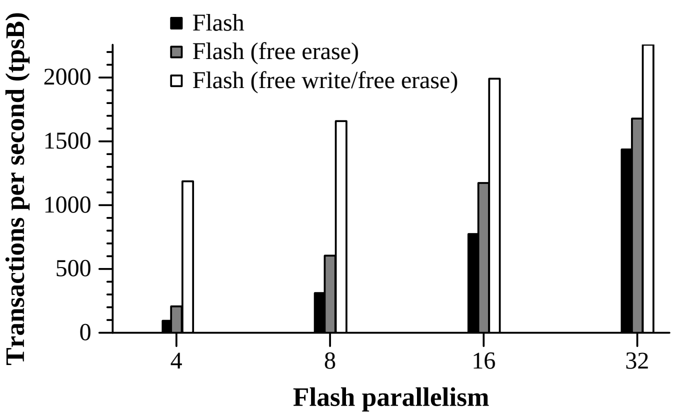

Figure 4: Gordon's TPC-B throughput

1000 bytes. If we recompute the lifetime for an actual ACARS groundstation load (less than 15 packets per second, with an average packet size under 100 bytes), we can expect almost a one hundred-fold decrease in the persistent GC frequency. In turn, this would correspond to a Gordon system lifetime of over 80 years!

8.3 Debit-Credit

The Debit-Credit benchmark was designed to test Gordon's high-performance transaction throughput. Figure 4 depicts our system's performance in terms of TPC-B transactions per second (tpsB). Even at the highest Flash parallelism modeled (32-way), Gordon is able to achieve only 1437 tpsB. Considering that high end database systems are more than an order of magnitude better [6, 13], Gordon is clearly not a suitable solution for heavy-duty transaction processing.

It is worth asking just how badly Gordon performs, however. Figure 4 also includes the tpsB measurements for Flash with zero-latency erases ("free erase"), as well as Flash with zero-latency erases and writes ("fast Flash"). It may seem odd that the fast Flash case is still affected by the level of parallelism, but this is easily explained by the fact that for lesser parallelism, our persistent heap size is smaller, so we have to perform persistent GC's more often. Since the x-axis is logarithmic, we can see that the fast Flash is beginning to level off at slightly higher than 2000 tpsB, indicating that we are CPU-bound at that point.

The most telling point, however, are the relative tpsB of Flash and free-erase Flash compared to fast Flash. Gordon is clearly held back primarily by erase bandwidth, and secondarily by write bandwidth, especially at lower levels of parallelism. Although the write and erase bandwidths scale linearly with the degree of parallelism, Flash still only performs at 67% of the rate of the CPU-bound case, even with 32-way Flash. At current prices, a 32-way Flash configuration would cost more than \$1300. Attempting to outperform high-end disk-based transaction systems by extending the parallelism even more is simply not cost-effective in today's market.

9 Related Work

The eNVy project [16] is exploring the use of Flash as main memory, essentially as a replacement for DRAM. They are implementing a memory controller that uses a large (2 gigabytes in their paper study), 256 byte wide Flash array together with battery-backed SRAM (64 megabytes) to provide the abstraction of in-place update on persistent memory. The hardware provides a program with the same interface as DRAM, except that writes persist even on power failure.

The simulations of their memory array suggest they can sustain write and erase rates sufficient to perform 30,000 transactions per second. A transaction system with the CPU overheads found here would need to have about 30 times the processing power of one of our R4400 CPUs to attain this rate. Such machines are available, although they must be considered much more high-end than the one used for our study.

In general, we believe our two approaches are quite complementary. Their approach provides high-end systems with a simple software interface, while ours allows for less expensive hardware but requires that the software respect the unique properties of Flash.

One interesting application that we are aware of is XIP, or eXecute In Place [7], which has been used in some low-power computers to execute the operating system out of Flash. XIP is special-purpose, while Gordon and eNVy are more general, but it still exploits the fact that Flash may be used for general-purpose, persistent immutable data quite readily.

Most other work on Flash has attempted to use it as a replacement for a disk, and in fact the commercial products currently available for general purpose computers provide either a file-system or a disk interface. These systems are usually targeted at mobile applications where the low power use and high ruggedness of Flash make its higher cost compared to disk less of a disadvantage. There is a small amount of literature devoted to this approach [5, 4], but it is targeted at a substantially different design point than the current work, and so is not particularly relevant.

10 Future Work

A number of optimizations to Gordon are possible, and we present some of them here as future work. Also, it would be illustrative to implement our interface using actual Flash hardware and a modified VM system.

10.1 Optimizations

In our current design we allocate all data into new space. If it lives long enough, it is copied into old space, and finally may be promoted to persistent space. The number of copy operations might be reduced by conservatively moving immutable data into the persistent heap directly from new space. This would essentially amortize part of the `commit` operation over the program execution improving overall latency. However, this advantage would have to be carefully weighed since our results indicate that Flash lifetime is extremely limited and may be severely reduced.

The speed of the garbage collector could be improved in a number of ways, such as through concurrency and better-remembered set representations; Sidney already supports these features, and it would be straightforward to transfer them to Gordon. However, since our results indicate that most operations are bounded by operations to Flash, it is not clear how much advantage this might provide.

A number of benchmarks were bounded by the rate at which Flash banks could be erased. This was determined to be at least in part because of poor persistent GC policy. It would be worthwhile to improve this policy as mentioned in section 8.2 to acheive 100% erase efficiency.

10.2 Actual Flash Implementation

An obvious next step is to implement the system with actual Flash. Because we already preserve Flash's semantics in software, we could modify a simple memory controller to support using Flash rather than DRAM. Conveniently, Intel supplies some Flash chips (the 28F016XD) with a DRAM like interface, and even has an application note explaining how to adapt existing DRAM SIMM implementations. This chip has similar performance to the one used here.

More aggressive designs are possible; for example, adding a tiny battery (or big capacitor) to the memory controller might allow it to use the page-buffers present on the 28F016SV for small contiguous writes, and guarantee persistence by forcing the data to Flash on power loss. This feature would almost triple the write bandwidth.

Real hardware will need to deal with issues involving the system's cache structure; in particular, writes would have to be forced all the way to Flash to persist, and erases would have to flush the cache to avoid reading stale memory. It seems likely that these issues can be dealt with using existing support for multiprocessors, although we have not investigated this question carefully.

Implementing the operating system interface that we've described appears to be straightforward, especially since the virtual memory system already provides most of the needed infrastructure.

11 Conclusions

The results of running our benchmarks are very favorable to Flash. Even for modest parallelism of Flash, Gordon is clearly superior to the disk-based Sidney. The largest deterrent to using Flash seems to be its cost, followed by limited lifetime and bandwidth. However, we have shown that for reasonably well-chosen applications, the latter two concerns are not limiting. Even for for the transaction processing benchmark, Gordon was only a respectable order of magnitude shy of typical high-end systems. For all of these reasons, we expect that Flash popularity will continue to grow, and may eventually overcome its cost limitation.

Acknowledgments

Thanks to Susan Davidson for the TPC benchmark pointers. We would also like to thank our readers, Scott Alexander, Alex Garthwaite, and Sanjay Udani. Finally, we would like to thank Alex Raymond for providing the inspiration for naming our system Gordon.

References

[1] Aircraft Communication Addressing and Reporting System. http://www.arinc.com/-Products_Services/acars.html.

[2] Flash Component Price List. http://-developer.intel.com/design/pricelst/ccom9.htm.

[3] Intel 16-Mbit FlashFile Memory. http://-developer.intel.com/design/flcomp/prodbref/-29734903.htm.

[4] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A Flash-Memory Based File System. In *USENIX Technical Conference*, 1995.

[5] F. Douglas, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage Alternatives for Mobile Computers. In *Operating Systems Design and Implementation*. ACM Press, 1994.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.

[7] Intel Corporation. *Flash Memory.* McGraw-Hill, 1996.

[8] Morgan Price and Scott Nettles. Transactions in a Flash. Technical Report XXX, Department of Computer Science, University of Pennsylvania, September 1996. http://www.cis.upenn.edu/-nettles/XXX.

[9] Samuel Dufour and Wink Saville. Advantages of Large Erase Blocks. Technical Report 297637-001, Intel Corporation, September 1995.

[10] Sanjay Udani. The Power Broker: Intelligent Power Management for Mobile Computers. Technical Report MS-CIS-96-12, Department of Computer Science, University of Pennsylvania, May 1996.

[11] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994. Corrigendum: *ACM Transactions on Computer Systems*, 12(2):165–172, May 1994. Also available in *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, December 1993.

[12] Scott Nettles. Safe and Efficient Persistent Heaps. Technical Report CMU-CS-TR-95-225, Carnegie Mellon School of Computer Science, December 1995.

[13] Transaction Processing Council. TPC Benchmark Results. March 7, 1997.

[14] Transactions Processing Council. TPC-B. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 79–114. Morgan-Kaufmann, 1991.

[15] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[16] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *Proc. 6th International Conference oa Architectural Support for Programming Languages and Operating Systems*, 1994.