Safe and Efficient Persistent Heaps

Scott M. Nettles

December 1995

CMU-CS-95-225

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee:

Jeannette Wing, Chair
Peter Lee
M. Satyanarayanan
Eliot Moss, University of Massachusetts

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the United States Government.

Keywords: Persistence, Transactions, Storage Management, Garbage Collection, Copying Garbage Collection, Concurrent Garbage Collection, Replicating Garbage Collection, Orthogonal Persistence

Contents

Acknowledgments

I am a very lucky person and much of my luck is in being fortunate enough to have many people and institutions to thank, in particular for their help in completing this dissertation and with it my PhD. No doubt I have forgotten many people; for this I apologize in advance.

Of course, my parents are responsible for much more than just my PhD. They have always been supportive and understanding even when they did not understand my somewhat tortured path. Without my mother's energy and drive for perfection, some of which I have inherited, this would be a different and lesser work. It might have been finished sooner as well! Without my father's perspective, it might not have been finished at all.

Long before arriving at CMU, many people played important roles in shaping me, my education, and my career. They include: Joe Gamble, Richard Schwendeman, Mike "Monty Python" Weaver, Brian Reid, and Forrest Basket. Without them who knows where I would be now.

The CMU School of Computer Science also deserves special mention. It is a special place, unlike any other graduate program I know of. It is not unique in the quality of education it provides, although this is superb. Where it really distinguishes itself is in the way it treats its students. I have benefited greatly from this treatment. This was why I came to CMU and I did not make a mistake. A few SCS people deserve to be named in particular, including Alan Perlis, Nico Habermann, Herb Simon, and Allen Newell. CMU computer facilities did an amazing job of keeping everything running. Special mention goes (as everyone knows) to Sharon Burks. When I first visited CMU, I was surprised that the only person my many CMU alumni friends wanted me to remember them to was Sharon. I now know why.

When I first arrived at CMU, a special bonus was the friendships I formed among my entering class. It meant a lot to make good friends so quickly and greatly eased the transition to living in Pittsburgh. Two members of my class stand out, Mike Gleicher and David Steere. Mike, Dave, and I have been through a lot together (and separately) and I could not ask for better "best" friends. They know how I feel and it is pointless to try and express it further in words more than thanks. David's wife, Jody, has also been a special friend; I thank her also.

Another group I found myself associated with early on was the Avalon group, with whom I started doing research upon my arrival at CMU. Dave Detlefs helped inspire my early interest in garbage collection and showed me that concurrent collection of persistent heaps was a tough problem and worth tackling. Upon hearing about replicating collection, he commented that it would make what was then his problem a lot easier. He was right. Stewart

Clamen help to fuel the Avalon/Lisp project, out of which many of my most interesting ideas flowed. Linda Leibengood was not only a good friend but a great co-worker. I have been poorer in all ways for her departure.

The Avalon group transmuted itself into the Venari group and it was in this context that most of my work at CMU was done. For me, it was always an effort to do Avalon/lisp right, although not everyone saw it that way. Venari group members include: Amy Moormann Zaremski, Darrell Kindred and Nick Haines (now Nick Barnes). Thanks to them. Special mention goes to Greg Morrisett, a great friend and great colleague. He and Jack know how I feel.

The Fox group has played an important role for me as well, as have all the people who go under the banner POP. Thanks especially to the Fox group's original PIs, Eric Cooper, Bob Harper, and Peter Lee, for helping to create an environment that is supportive of doing systems work in reasonable languages. Special thanks to Peter Lee for being a friend and mentor and for his excellent teaching.

The Coda group also deserves thanks both for building a great program to benchmark and for work relating to RVM. Particular thanks to Satya for all of his help, especially reading my dissertation quickly in a pinch. Thanks to David Steere for his work on RDS and to Puneet Kumar for providing his TPCB benchmark.

Thanks to all the research groups that have provided such an active intellectual life at CMU, including POP, the hardware lunch, and the systems lunch. The pizza helped me not to starve as well. Thanks to all my fellow graduate students for friendships of all kinds. It's been fun. Drop by sometime.

Jim O'Toole has played a pivotal role in this work, both as the co-inventor of replicating collection and as my partner in crime for much of the work presented here. It's difficult to imagine a more pleasant, fun, or productive partnership, in which shared attitudes and complementary talents come together. Now that we are both finishing, I look forward to getting back to it! Thanks also to Jim's group for allowing me the extensive use of its SGI multi-processor.

I met Jim as a summer student at Digital Equipment Corporation's Systems Research Center. Thanks to DEC and especially to Jim Horning for this opportunity. Without it my friendship/partnership with Jim would have been impossible.

Music has always been a love and it has been a major help it getting through the last few, especially difficult years. Thanks to Aretha, The Meters, George Clinton, Bootsie and the rest of the P Funk gang, Maceo, Fred, and Pee Wee, Al Green, the DJs at WAMO and Marvin Gaye. A special shout out to James Brown, the godfather of soul and original funk master. Hit me!

So many friends deserve mention that to enumerate their contributions in detail would be impossible. They include: Penny Anderson, Francesmary Modugno, Juan Leon, James Landay, Lin Chase, Jay "Sippy" Sipelstein, Brian Noble, Kathy Swedlow, The Zephyr Dinner Group, Cathy McGrath, Phyllis Reuther, Gene Rollins, Qingming Ma, Spot, Amy Merrit, Nix, Ellen Seigel, Lori Fabrizio, Morgan Price, Puneet Kumar, Jay Kistler, and Stewart Clamen. Special thanks to James for those late night stress relieving hot tub sessions. I wish I were soaking in it with "a cold one" now.

I had imagined that my leaving CMU and transition to a new life would be a quick thing, but it was not. Instead, my new life in Philadelphia at Penn and my old one in Pittsburgh at CMU have blended in a way I can only term "interesting". I am lucky to find my new environment as supportive (if somewhat less organized) as my previous one. Thanks to Carl Gunter for his part in bringing me here. Extra special thanks goes to Jonathan Smith, for service above and beyond the call of duty, especially in helping me make time to get this document finished. One cannot ask for a better colleague. Luck strikes once again!

For all the pain of the last year of my PhD and the simultaneous first year at Penn, one thing has always been a pure joy. That thing was growing to know and to love, Tandy Warnow. She has been kind and patient and understanding thorough long difficulty. She even read my drafts. I look very forward to a long life together.

In the process of finishing I found myself in Pittsburgh frequently. Thanks to those who gave me a temporary home, Greg and Jack, Nix, and Dave and Jody. Thanks also to Darrell Kindred for office space and a home for my machine, ldl.venari.

My committee, Peter Lee, Satya, and Eliot Moss, also deserve great thanks. It's hard to believe they read the whole thing! Special thanks goes to Eliot for his expert help and positive feedback throughout the process, as well as for a willingness to change travel plans.

Jeannette Wing, my advisor, deserves more credit than any other for this work. Jeannette gives of herself to her students more than any advisor I have ever encountered; it is impossible to express my gratitude or to explain the ways she has contributed to me, my education, and my work. I always tell new graduate students that one's advisor is the most important factor in being a successful graduate student. Choosing Jeannette was a grand slam for me. I can imagine only one way to repay her and that is to pass the favor along. In everything Jeannette sets a high standard and I have always struggled to meet it. I hope that as an advisor and for my students I can manage to do a small fraction of what she has done. Thanks is too little to offer.

Abstract

Persistent data continue to exist after the programs that create or manipulate them terminate. Files and database records are familiar examples, but they only allow specific datatypes to be made persistent. Object-oriented databases and persistent programming languages are examples of systems that require arbitrary persistent datatypes. Persistent heaps allow arbitrary persistent datatypes to be dynamically allocated and stored and are essential components of such systems.

Data stored in persistent heaps are valuable, and must be protected from both machine failures and programmer errors. This safety requirement may conflict with the need to provide high throughput and low latency access to the data. This conflict may lead to sacrificing safety for performance.

My thesis is that it is possible to build persistent heaps so that safety does not need to be sacrificed for performance. This dissertation demonstrates the thesis in two parts: Part I presents the design of Sidney, a safe persistent heap, along with the details of its implementation, and Part II presents a performance evaluation that demonstrates that Sidney satisfies the claim of the thesis.

Sidney's design uses transactions and garbage collection to provide safe heap management of persistent data. Good performance is achieved by combining traditional systems techniques for transactions with a novel concurrent garbage collection technique, replicating collection.

Sidney's implementation is the first to provide concurrent collection of a transactional heap. Replicating collection allows a much simpler implementation than previous (unimplemented) designs based on other concurrent collection techniques.

The performance evaluation characterizes Sidney's performance and compares it to other approaches, including a persistent malloc-and-free implementation. It shows that replicating collection allows the use of garbage collection without sacrificing the throughput and latency characteristics of explicit deallocation. In fact, not only does the Sidney provide better safety than persistent malloc-and-free, it also provides better performance.

List of Figures

Part I

Design and Implementation

Chapter 1

Introduction

The uses of computer systems would be greatly limited if it were impossible to store information permanently. Persistent data are permanent and continue to exist after the programs that created them are no longer running. Persistent data provide the means by which programs create and manipulate permanent information; files and database records are familiar examples.

Both files and databases make very specific data types (arrays of bytes and relations, respectively) persistent and to use them programmers must map all data types in their programs to these specific types. In contrast, systems that support *general-purpose persistence* allow arbitrary data types to be made persistent. Examples of such systems include object-oriented databases [9, 11], persistent programming languages [5, 38, 62], and general-purpose transaction systems [53, 57]. Such systems are the focus of this dissertation.

Persistent data are not destroyed by program or machine failures. When a program recovers from a failure, the question is, "In what state is the persistent data?" It is important that the state not be inconsistent. For example, in a bank transfer either both accounts must be updated or neither account should be updated. Having money withdrawn from one account, but not deposited in the other is unacceptable. *Transactions* are the basic mechanism by which programmers can control the recovery state. Transactions allow changes to persistent data to be grouped into atomic units that are either done completely or not done at all. This all-or-nothing property of transactions allows the programmer to guarantee that a system is recovered to a consistent state.

Support for general-purpose persistence requires that it be possible to allocate persistent storage dynamically, in much the same way that file systems must allow new files to be created dynamically. To support dynamic storage allocation, we use a data structure known as a *heap*. Heaps allow the programmer to allocate new data as well as to examine and to modify existing data. Some heap implementations also require the programmer explicitly to deallocate data that are no longer in use; other systems do not make this requirement, but instead deallocate storage automatically or implicitly. Implicit storage deallocation is also known as *garbage collection* (GC), because the storage that is deallocated must not be useful: it is garbage.

My thesis is:

> A persistent storage manager based on implicit storage management techniques is safer than one based on explicit storage management techniques; furthermore, such a system can have performance that is competitive with those based on explicit techniques.

Sidney, the system reported in this dissertation, provides dynamic allocation of general-purpose persistent data and transactional modification of that data. The design and implementation of Sidney emphasize both **safe** and **efficient** manipulation of persistent storage. Crucial to Sidney's safety goals is the use of garbage collection to deallocate unused storage safely. Crucial to Sidney's performance goals is the use of a new concurrent garbage collection technique, *concurrent replicating collection*, to deallocate unused storage without the performance disadvantages of conventional collectors. Sidney is the first successful implementation of a concurrent collector for a transactional persistent heap; replicating collection plays an important role in this achievement. Although I use the term "Sidney" to refer to both the design and implementation presented here, the design transcends any particular implementation; the implementation serves to demonstrate that the design is implementable, as well as forming a concrete basis for evaluating the performance characteristic of the design.

This chapter provides an overview of the key concepts underlying Sidney's design. It begins by considering Sidney's basic design goal, achieving safe management of persistent data, and how Sidney achieves this goal. It continues by considering Sidney's performance goals and how they are achieved. It then reviews Sidney's basic design, followed by a review of the contributions of the thesis. A guide to the rest of the thesis is found at the end of the chapter.

1.1 Safety

Persistent data are valuable; if they were not, there would be no reason to make such data permanent and to protect them from program failures. Since persistent data are valuable, it is important that the management of persistent storage be as safe as possible. Common programming errors should not threaten the integrity of persistent data.

The most important design goal for Sidney is to provide safe storage management for persistent data. A secondary goal is to provide a system that is well integrated with the programming language forming the basis of the implementation and that is convenient for the programmer to use. In many cases, there is a choice between using techniques that require explicit action by the programmer and those in which the system implicitly performs these actions. Since they tend to be safer and more convenient than explicit techniques, in general, Sidney uses the implicit techniques.

These features are the primary contributors to safe storage management:

- Transactions: Support for atomic update of persistent data.
- Garbage Collection: Support for implicit storage deallocation.
- Orthogonal Persistence: Support for implicit determination of persistence.

The following parts take a closer look at how each of these features contributes to safe storage management in Sidney.

1.1.1 Transactions

After the basic choice to use general-purpose persistence, the single most important design decision in Sidney was to support transactions. Despite the emphasis in Sidney on implicit techniques, I chose to support transactions because, for many applications, programmers must have explicit control of the state that is recovered after a failure. Transactions are the main explicit technique in general use for controlling the recovery state and supporting fault tolerance. Since it is desirable to allow the programmer to have guarantees about even small changes to persistent data, there is a further emphasis on efficient support for small transactions that only modify a small amount of state.

Transactions are a control construct that supports the consistent modification of persistent data. A transaction may complete successfully and *commit* its results, making them permanent, or it may fail and *abort*, rolling back any changes made during the transaction. Traditionally, transactions have the following properties:

- Durability: If a transaction commits, any modifications it has made to persistent data are made permanent and will not be lost, even in the event of a system failure.
- Atomicity: The effects of a transaction are “all-or-nothing.” Either all of its effects take place or none of them do.
- Serializability: If multiple transactions execute simultaneously, then the effects on the overall state of the system are as if they executed in some serial order.

Transactions support the safe manipulation of data by allowing operations to be grouped into atomic units that are performed all-or-nothing. Transactions prevent intermediate, inconsistent states of the system from being observed on recovery, because these states are never made permanent. If the system crashes and the state of the system must be recovered, the user need not worry whether the data are inconsistent. Knowing that the system recovers from failure in a consistent state greatly simplifies building fault-tolerant applications.

Since the transaction system must make any modifications to data permanent on commit, it must somehow be able to track these modifications. Sidney does not require the user to record changes explicitly; instead it implicitly tracks all changes. This technique enhances safety because there is no possibility that the user will forget to record a modification.

1.1.2 Garbage Collection

Any system that supports dynamically allocated storage must also provide some mechanism to reclaim storage that is no longer used. Since it is impossible to reclaim the storage simply by exiting the program, this requirement is particularly essential for persistent heaps. It is possible for storage deallocation to be either explicit or implicit, and many systems have been built using one approach or the other.

The key problems with explicit deallocation are safety-related. Programmers may neglect to free storage that is not being used, leading to a *storage leak*. Leaks are a particular problem for persistent heaps since unreclaimed storage persists as well. Leaks can compromise system integrity by leading to crashes due to unbounded memory use. The second problem is more severe. Programmers may accidentally releases storage that is still in use; the storage manager will eventually reallocate this storage, at which point the storage is being used inconsistently for completely different purposes. Such unintentional aliasing of storage compromises data integrity and can lead to errors which are very difficult to debug. Given the valuable nature of persistent data, this problem is particularly severe.

Sidney uses garbage collection to reclaim unused storage. Garbage collection does not suffer from the preceding problems because programmers do not free storage themselves. Instead, the garbage collector identifies the storage in use and frees any unused storage, without programmer intervention.

To find the usable storage, garbage collectors start from a set of locations, called the *roots*, that are known to be usable. The collector then finds all the data that are reachable by dereferencing pointers starting from the roots; these data are called *live*. Any data that are not live are garbage and can be reclaimed.

1.1.3 Orthogonal Persistence

If performance and cost were not an issue, all data could be made persistent. In practice, persistent data are more expensive to manage than transitory data. Thus, practical systems that support persistent data must also provide support for transitory data as well.

If data can be either transitory or persistent, there must be some way to determine which each datum is. Again, there are two basic approaches, explicit and implicit. Like garbage collection, the implicit approach, *orthogonal persistence*, is safer and arguably more convenient for the programmer [5]. In a system that supports orthogonal persistence, there is a persistent root. Any datum that is reachable by dereferencing pointers from the persistent root is itself persistent.

Orthogonal persistence is safer than explicit approaches because it avoids the possibility of dangling pointers if the system crashes and recovers. The issue is whether or not there can be pointers in the persistent heap that refer to data in the transitory heap. If there are and they are saved to permanent storage, then on recovery they will be invalid, since they point to data that no longer exist. In a system supporting orthogonal persistence, no such pointers may exist, since all data reachable from the persistent roots are persistent and therefore part of the persistent heap. Since orthogonal persistence frees the programmer from having to decide whether data are or are not persistent, it is also easier to use.

1.2 Performance

A system that is safe but has unacceptable performance will go unused in favor of one that is less safe but has acceptable performance. Thus, the most important secondary design goal for Sidney is to provide good performance. Since the primary goal, safety, dictates the use of implicit storage management, the secondary goal requires that these implicit techniques have good performance. Safety is inherent in the basic design of Sidney, but good performance is achieved by the careful engineering of its implementation.

For each contributor to Sidney's safety properties, there is a corresponding contribution to achieving good performance, as follows:

- Transactions: Caching data in volatile memory and logging for fast disk update.
- Garbage Collection: Concurrent replicating collection to bound GC pauses and to increase throughput by exploiting concurrency.
- Orthogonal Persistence: Replication-based rollback to avoid long commit latencies.

The following parts take a closer look at how each of these features contribute to good performance in Sidney.

1.2.1 Transactions

Currently, the commonly available permanent or *stable* storage is disk-based. Disks are slow, and accessing persistent data directly on disk is unlikely to have acceptable performance. Also, transactions require that changes to persistent data be made atomically. In Sidney, both of these issues are addressed by caching a copy of persistent data in fast volatile memory and making all client accesses and modifications there. Caching allows the most common operations, `read` and `write`, to operate at normal speeds. Atomicity is easier to achieve because the primary version is not changed by the user.

When a transaction commits, any persistent data that have been modified must be transferred to stable storage atomically. However, it is possible and even likely that changes have been made to scattered locations throughout the heap. Writing these changes to scattered locations on disk is likely to be expensive because it would require many seeks, and each seek may also incur some rotational delay as well; furthermore, multiple disk updates are also hard to make atomic. The technique used by Sidney is the conventional one: rather than writing the changes to scattered locations, all of the changes are written to the end of a disk-based log. Using a log avoids multiple disk writes, and if the disk arm has not been moved, may also avoid seeks.

For Sidney, disk logging services are actually provided by another subsystem, Recoverable Virtual Memory (RVM). RVM allows portions of files to be mapped in memory, and provides efficient logged-based update of the disk. To provide these services, RVM must be notified of the locations of parts of memory that have changed.

Sidney supplements RVM by implicitly recording any `writes` in a *write log*. When the disk must be updated, Sidney informs RVM of any locations that have changed. Sidney makes recording the changes fast by using in-line code emitted by the compiler to log the `writes`. As we will see, this write log plays a pivotal role in most of Sidney's implementation.

1.2.2 Garbage Collection

The most important performance problem with garbage collection comes because basic garbage collectors stop the client while reclaiming storage. The length of the resulting *GC pause* is proportional to the amount of useful storage in the heap. The resulting pauses can be seconds or even minutes long, and thus are unacceptably disruptive to many applications.

To solve the problem of GC pauses, Sidney uses a novel concurrent garbage collection technique, *replicating collection*. Replicating collection allows the client and user to execute concurrently, and limits the GC pauses to a brief period when the collection terminates. Concurrent collection also improves the throughput of the system since it allows the client and collector to overlap their execution. Since Sidney is I/O intensive, this improvement is significant even on uniprocessors, due to overlapping execution with I/O.

1.2.3 Orthogonal Persistence

In a system that supports orthogonal persistence, all data reachable from the persistent root are persistent and must reside in the persistent heap. Since in a transactional system only the last committed state will be observed if a crash occurs, this requirement only needs to be enforced when there is a commit. On a commit, Sidney uses the write log to identify pointers that point from the persistent heap into the transitory heap, and then uses garbage collection-based techniques to move the data reachable from these pointers into the persistent heap.

This basic strategy works well, except that it leaves pointers in the transitory heap that point to versions of data that have been moved into the persistent heap. Correcting these pointers to refer to the persistent heap versions takes time proportional to the amount of data in the transitory heap. The need to correct these pointers means that the time needed for a commit can be unacceptably long if there is a substantial amount of transitory data.

Sidney solves this problem by using a technique based on replicating collection called *replication-based rollback*. Replication-based rollback avoids the need to redirect the transitory heap pointers and allows commit to have the best asymptotic complexity possible.

1.3 Design Overview

In this section, I present a brief overview of the design of Sidney, starting with the basic system interface. This overview follows the structure of O'Toole, Nettles, and Gifford [44] closely but it is less detailed. The next three chapters provides a detailed picture of the design, but following a somewhat different breakdown than used here. The implementation details are found in Chapter 5.

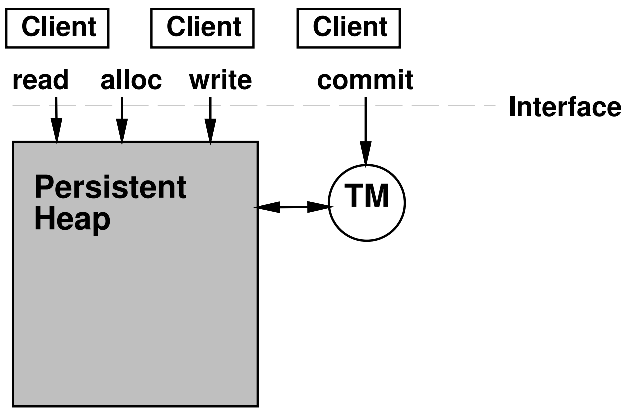


Figure 1.1: The Transactional Heap Interface

1.3.1 System Assumptions

Sidney is not a system built in a vacuum and its design is based on assumptions and biases contributed by its context. In particular, Sidney was designed to be the low-level support for a transaction system for Standard ML (SML) and in particular for the Standard ML of New Jersey (SML/NJ) implementation. Garbage collection and orthogonal persistence were provided to support the SML programmers' high-level view of memory. Safe storage management would not be something one would sacrifice for performance when extending SML.

The implementation of SML/NJ also had its influences. The collector used by SML/NJ is a copying collector with a simple generational collector. SML/NJ implicitly logs some writes to support the generational collector, making it natural to extend this log for use by the transaction system. Orthogonal persistence fits very well with GC, and the idea of using a copying collector to identify newly persistent data and transport them to the persistent region of memory seems an especially natural one. Despite Sidney's origins in the SML/NJ implementation, the basic design should apply to any programming language implementation that supports copying collection.

There is a cost to using copying GC. In the most simplistic form, as used by Sidney, the two-space nature means that when collecting, the memory use can be double that of the client alone. Generational techniques may make this less of a problem, but it remains the case that one of the costs of GC is likely to be greater memory use.

1.3.2 System Interface

Figure 1.1 shows the basic system interface. Persistent data are stored in a *transactional persistent heap*, which is located in stable storage, indicated by the gray in the figure. The basic heap operations are `read`, `alloc`, and `write`. The transaction operation is `commit`. Transaction abort is also supported. I do not include it in the interface because I only discuss it when motivating certain implementation details. The transaction manager (TM)

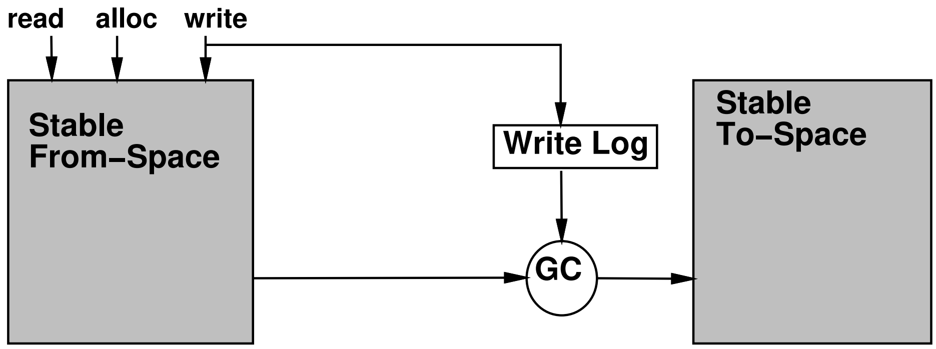


Figure 1.2: Replicating GC of a Persistent Heap

represents the portion of the system that implements the transaction semantics. The lack of an operation that can release storage that is no longer needed means that the system must use GC. The multiple clients indicate that Sidney's design (although not the current implementation) can support multiple concurrent transactions.

Sidney is part of a larger system that supports all of the aspects of transactions [23], not just persistence. However, in the interests of clarity and focus, I will concentrate on the aspects that provide durability and atomicity with respect to system failures, and I will note when alternative designs or implementations would limit the ability to implement `abort` and multiple or nested transactions.

1.3.3 A Garbage-Collected Persistent Heap

Figure 1.2 shows a block diagram of a basic garbage collected persistent heap. This design does not support transactions. The client `reads`, `writes`, and `allocs` data directly in stable storage. When a garbage collection is required, a concurrent replicating collector is used to copy the stable from-space directly to the stable to-space. Since disks are used to implement the stable spaces, the performance of this design would likely be unacceptable.

As shown, a copying collector copies data from from-space to to-space. Only data that are reachable from the roots by dereferencing pointers are copied, all other data in from-space are garbage. The usual technique for copying the data overwrites the from-space data with *forwarding* pointers that allow the collector to find the to-space version of a datum given a pointer to the from-space version. When the collection is complete, a *flip* is done and to-space becomes from-space. The old from-space is then reused.

The most basic copying collectors, *stop-and-copy* collectors, stop the client from executing when a collection is needed, and do not allow it to resume until all of the copying is done and the flip is complete. This creates a GC pause whose length is proportional to the amount of data copied. Unfortunately, many applications cannot tolerate these pauses, which limits the usefulness of stop-and-copy collection.

To avoid having GC pauses that depend on the amount of data in from-space, it is necessary for the collector to be able to copy data concurrently with the clients' execution. In Sidney this is achieved by using concurrent replicating collection. When a collection

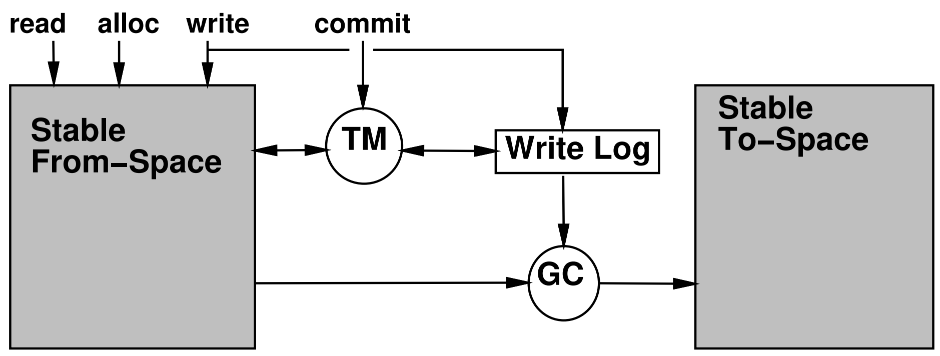


Figure 1.3: A Transactional Persistent Heap

is needed, Sidney starts the garbage collector, which runs concurrently with the client. Eventually, the collector finishes all of the copying; it then interrupts the client and performs the flip. At this point the collector terminates until the next collection is needed.

While the collector creates copies or *replicas* of data in to-space, replicating collection requires that the client continue to access the data in from-space. This requirement is called the *from-space invariant*, and it implies that the collector cannot destroy the from-space version by overwriting it with a forwarding pointer.

The user may `write` the from-space version of an object after the collector has copied it. However, when the collection completes, the to-space version must be the same as the from-space version. Replicating collection records all `writes` in a write log, so that before a flip occurs they can be applied to the to-space versions. Thus, before the flip can occur, all the data must be copied into to-space, and any changes recorded in the write log must be applied to the to-space versions.

1.3.4 Adding Transactions

Figure 1.3 shows a new design that supports transactions. A new operation has been added, `commit`. `Commit` makes any current modifications permanent, while `abort` (not shown) undoes or rolls back any current modifications.

Since `writes` and `allocs` are done directly to the stable from-space, to support `abort` the write log must also contain the old values of overwritten values, as well as the locations of the `writes`. The transaction manager is responsible for actually implementing `abort` and `commit`. When an `abort` occurs, the transaction manager reads the write log and uses the old values found there to rollback any changes to the heap. `Commit` simply discards the write log, making all of the changes permanent. The garbage collector continues to work as before.

1.3.5 Using Volatile Memory

Although the previous design supports most of Sidney's desired semantics, lacking only support for transitory data, its performance will be limited by the speed with which the disk

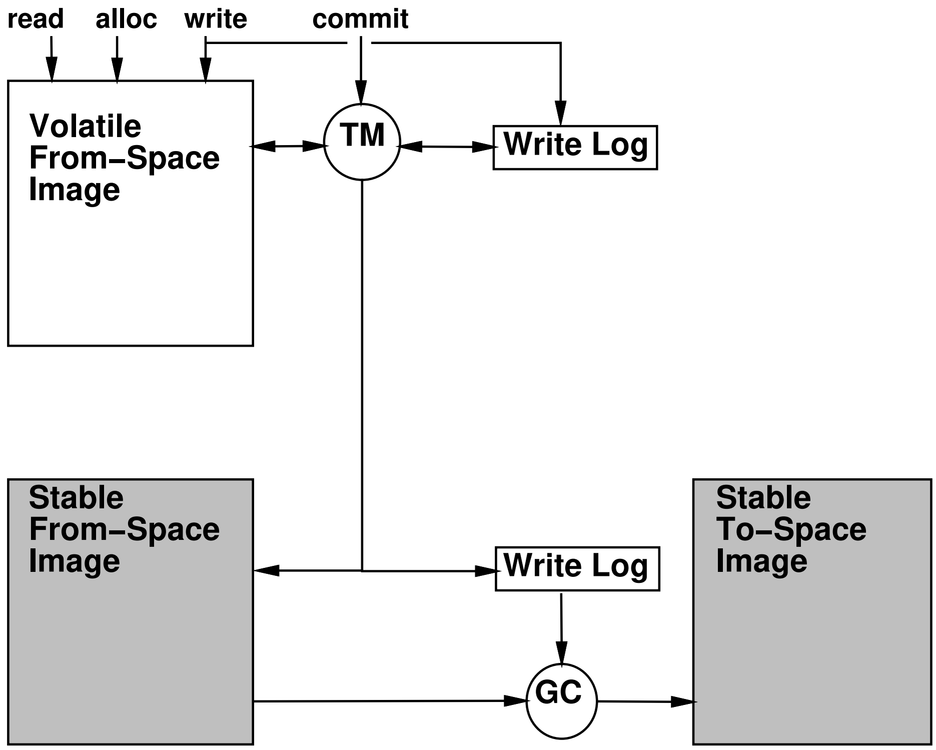


Figure 1.4: Using Volatile Memory

can be accessed. An obvious solution is to cache a copy of the data in volatile memory.

Figure 1.4 shows a version of the design in which the client `reads`, `writes`, and `allocs` data in volatile memory (shown as white) but the garbage collector continues to work on the stable versions of the heap.

Now, no changes are made directly to the committed version of the data on stable storage. Instead, `commit` must transfer any changes from the volatile from-space to the stable from-space. The write log captures the modified locations, so the transaction manager can just traverse this log and write the changes to disk. The transaction manager must also inform the collector of what changes have been made, so that the collector may keep the stable to-space up-to date. The write log connected to the transaction manager refers only to uncommitted data, while the one connected to the garbage collector refers only to committed data.

`Abort` can be implemented either by making the write log contain old values and thus act as an undo log, or simply by replacing the modified volatile from-space with the stable from-space. Since the garbage collector acts only on the stable images, when it flips, it must replace the volatile from-space with a copy of the stable to-space.

1.3.6 Improving Garbage Collector Performance

The previous design will have poor collector performance because the collector still acts directly on stable storage. The solution is to collect the volatile cache introduced in the

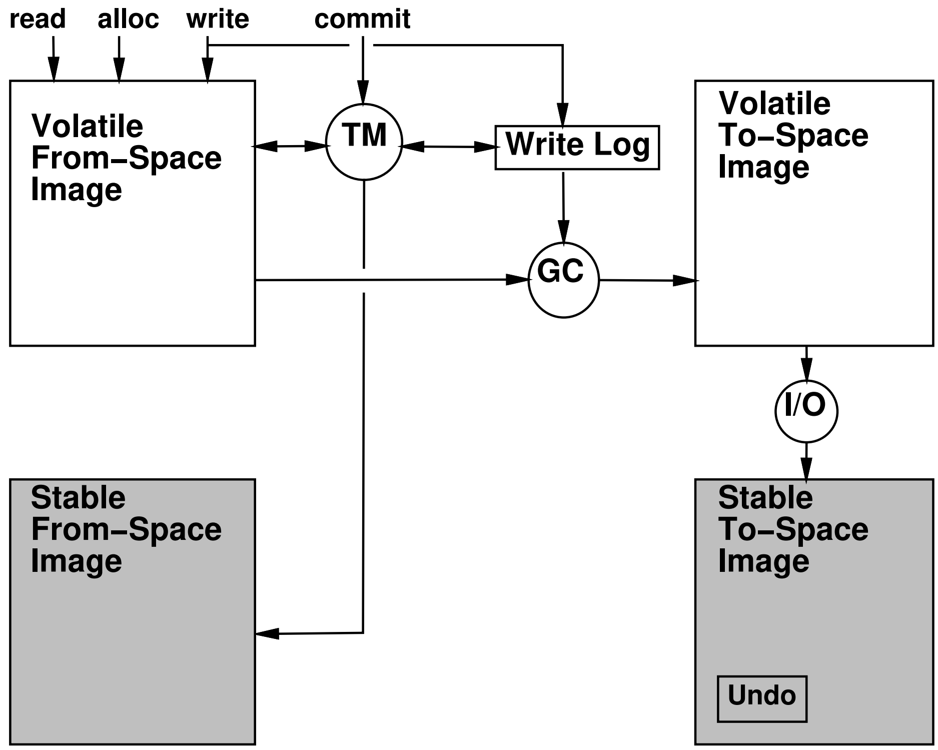


Figure 1.5: Improving Garbage Collector Performance

previous design into a volatile to-space. Before the collection can complete, this volatile to-space must be written to stable storage.

Figure 1.5 shows the new design. The I/O thread has been added; it writes the volatile to-space to stable to-space. This copying can be done concurrently with the collection.

A new problem arises because the volatile to-space contains uncommitted data. The collector will copy these data into volatile to-space, and from there the I/O thread will copy them onto stable storage. This means that the stable to-space is not a valid copy of the stable from-space, since it will contain uncommitted data. The solution, shown in the figure, is to include the undo log in the stable to-space. If the system crashes, the recovery process can use the undo log to restore the heap to its committed state.

1.3.7 Adding a Transitory Heap

The design still has one problem: all data are persistent. Figure 1.6 shows the system with the addition of a transitory heap; all of the spaces of the previous design are now grouped together as the persistent heap. The system now has two root sets, one for transitory data and one for persistent data. The system supports orthogonal persistence, which means that any data that are reachable from the persistent root must be persistent. This requirement is enforced by `commit`, as shown in the diagram by the fact that `commit` may move data from the transitory heap into the persistent one. Since it is essentially impossible to tell at allocation time if data are persistent or not, all `allocs` occur in the transitory heap.

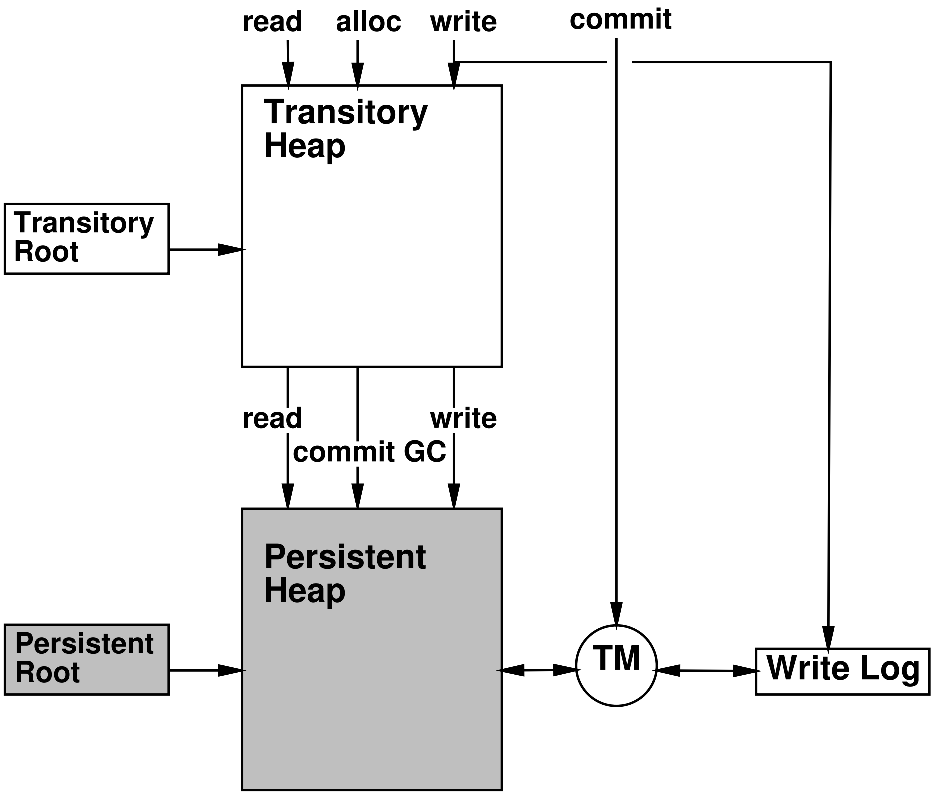


Figure 1.6: Using a Transitory Heap

1.4 Main Contributions of the Dissertation

The goal of this dissertation is to show that it is possible to support a transaction-based general-purpose persistence system that is not only safe, but also provides performance that is competitive with less safe approaches. To meet this goal, I took three conceptual steps. First, I designed a general-purpose persistence system that is safe and has asymptotic performance that is competitive; this showed that the basic goal was not impossible to achieve. Second, I implemented this design; this shows that the design can be realized in practice. Finally, I carefully benchmarked this implementation, along with examples of alternative approaches for comparative purposes; this shows that the implementation meets the goals of the dissertation.

The most important contributions of the dissertation are:

- A new technique for concurrent copying garbage collection, replicating collection, that allows the use of a from-space invariant.
- A new design of a concurrent collector for a transactional persistent heap that improves on previous designs, in part because of simplifications due to the use of replicating collection and a from-space invariant.
- The first implementation of a concurrent collector for a transactional persistent heap.

- The first design and implementation of an orthogonal persistence system that supports transactions and has optimal asymptotic complexity.

- A performance evaluation that compares implicit persistent storage management to explicit persistent storage management and shows that the implicit techniques can achieve performance that is competitive with the explicit ones.

A more detailed list of contributions is found in the conclusions, Chapter 12, where they may be more easily understood.

Although the above are the high-level contributions of the work described in this dissertation, I should make it clear that not all of these are my individual contributions. First, Sidney itself grew out of the Venari project at CMU, led by Jeannette Wing. Second, my most direct collaborator, Jim O'Toole of MIT, made many substantial contributions; the most important of which is as the co-developer of replicating collection. We shared the responsibilities for the first implementations of replicating collection for both transitory and persistent heaps, which used the SGI multiprocessor. I ported this implementation to the DEC uniprocessor. O'Toole implemented the versions of TPC-B and OO1 in SML, and together we designed the first versions of the **trans**, **comp**, and **OO1** benchmarks.

1.5 Roadmap

The dissertation is divided into two major parts: Part I, design and implementation, and Part II, performance evaluation. This chapter serves as an overview of the design and implementation. Chapter 6 serves as an overview of the performance evaluation. It should be possible to get a basic understanding of the dissertation by reading these two chapters alone.

The design and implementation part is organized in the following way. Chapter 1 is this overview; it presents the motivation and goals of Sidney, followed by a brief overview of the design, and concludes with the contributions of the dissertation. Chapter 2 presents the design of the transaction system. Chapter 3 presents the design of the garbage collector. Chapter 4 presents the design of support for orthogonal persistence. Each of these design chapters follows the same structure. They begin with a more detailed motivation, followed by the basic design, and end by looking at some design alternatives. Chapter 5 describes the implementation in considerable detail; it concludes the design and implementation part.

The performance evaluation part is organized in the following way. Chapter 6 presents an overview of the performance evaluation; it begins by providing the motivation of and justification for the benchmarking strategy, followed by the high-level results of the benchmarks, and ends with a discussion of the benchmarking methodology. Chapter 7 presents the detailed results of the **trans** benchmark. Chapter 8 presents the detailed results of the **coda** benchmark. Chapter 9 presents the detailed results of the **comp** benchmark. Chapter 10 presents the detailed results of the **OO1** benchmark. Each of these detailed benchmark chapters are organized in the same way. They begin by motivating the choice of benchmark, followed by a description of the benchmark itself. They then present the detailed results of running the benchmark, starting with the throughput-related results and followed by the latency-related results. Chapter 8 does not include any latency results.

Chapter 11 presents related work. Finally, Chapter 12 presents a more detailed list of contributions, a summary of the dissertation, some suggestions for others, and some proposed future work.

Chapter 2

Transactions

The fundamental goal of Sidney is to support the safe, reliable use of persistent data. Support for transactions is central to meeting this goal. As originally formulated, transactions combine features for managing concurrency, rolling back side effects, and making multiple updates to persistent data atomic. In this dissertation, I focus on the aspects of transactions that concern atomically updating persistent data, but the techniques presented are compatible with all of the standard semantic features of transactions.

Central to understanding transactions is the issue of failure recovery, or simply *recovery*. When a system fails, it is important to know what the state of the system is when it is restarted or recovered. Allowing the programmer to control the recovery state is crucial since many modifications to data leave a system in an inconsistent state. Transactions help solve this problem by allowing the programmer to control when groups of updates change the recovery state. Transactions support atomic update of persistent data through the `commit` operation. When a `commit` is performed, all changes to persistent data are transferred *atomically* to stable storage, i.e., as a single unit. If a system failure occurs before the `commit` completes, the persistent data remain in the state that they were in before the `commit`. If the `commit` succeeds then all of the changes to persistent data are permanent. Thus, in a transaction-based system, determining the state to which data are recovered is simple: recovery puts the system in the last committed state. These semantics provides the programmer with very concrete guarantees about what happens in the event of a failure.

Without these kinds of guarantees, the programmer will have to deal with the possibility that the recovered state is inconsistent. Doing so can be extremely difficult and essentially requires the user to create their own special-purpose transactions, since they must make sure that all possible states are ones that are acceptable to recover to.

Systems that manipulate persistent data without transactions can also be grossly inefficient. For example, in the BSD 4.2 file system, certain file system updates require seven synchronous disk writes, each of which is expensive [48]. If the system were based on transactions, it would be possible to perform the same update using one synchronous write. That is not to say that transactions themselves are cheap; with current technology `commit` requires an expensive synchronous disk write. But any system that must make guarantees about the state of data on disk will require such writes; so, at least in terms of the number of synchronous disk writes, transactions are no worse than other possible solutions.

This chapter begins by discussing the basic issue of how Sidney supports both safety and performance in its transaction design. The chapter proceeds with a discussion of the design at a relatively high-level. The chapter concludes with a discussion of design alternatives.

2.1 Safety and Performance

In a transaction-based system, the programmer can safely assume that on recovery the system is in its last committed state. Thus transactions are fundamentally concerned with providing safety guarantees to the programmer. The fundamental challenge that transactions present is how to implement them efficiently.

Stable or persistent storage is storage that is not destroyed by system failures. System failures include the unexpected power failure and loss of all *volatile* memory in the system. In typical current systems the stable storage that provides the lowest latency and highest bandwidth access at acceptable cost is magnetic disk. Failures of disks themselves are outside of the scope of this dissertation. Disk latencies are on the order of tens of milliseconds. Programs generally manipulate data in volatile semiconductor memory, which has access times on the order of one hundred nanoseconds, or CPU registers, which have access times in the general range of ten nanoseconds or less. Overcoming the performance gap between volatile and stable storage is the primary performance challenge in implementing transaction-based systems.

Sidney uses a standard technique to overcome the difference in performance of disks and memory by caching a copy of the persistent data in volatile memory. Caching allows `reads` and `writes` to persistent data to proceed at volatile memory speeds. Using a copy of the data also helps in providing atomicity, since the original data are not being changed. With this technique, `commit` must transfer any changes made to the volatile version of the data to the stable version. It is this transfer of data that must be made atomic.

An important design goal is to provide efficient support for `commit` when only a small amount of data have been modified. Meeting this goal requires that the system be able to find what data have been modified, so that it may transfer a limited amount of data to disk. The need to track what data has been modified raises a safety-related design choice. Should the system require the programmer to notify it explicitly when `writes` occur, or should the system keep track of `writes` implicitly without the need for programmer involvement? If the programmer must explicitly notify the system about `writes`, there is a distinct possibility that the programmer will fail to do so. If the system does not know that some location has been overwritten, it will be impossible for it to correctly update the stable version of the data and inconsistencies may arise. Sidney tracks `writes` implicitly, guaranteeing that there are no lost updates.

The basic technique used to make updates to disk both atomic and (relatively) efficient is to write updates to the end of a disk resident log. Logging achieves atomicity because in addition to the records describing the updates themselves, the system can include special log records to mark atomic sequences. If a crash occurs, the system can then use these records to reconstruct the final committed state. Logging helps provide efficient disk update because instead of requiring many writes to different parts of the disk to update stable storage, a single log write can be used. Furthermore, if the disk arm has not been moved since the last log write, this write will not require a seek. Dedicating a disk to the log or placing the log on an otherwise infrequently used device will allow most log writes to avoid seeks. Sidney's disk-based log is actually managed by a separate subsystem, Recoverable Virtual Memory (RVM). Details of how RVM manages the disk log and how it is used in recovery can be found in Satyanarayanan, et al. [51]; they are not relevant to the rest of the dissertation.

2.2 Design

This section discusses the basic design of transaction support in Sidney. Most of the approaches taken here are not novel: many transaction systems face the same set of issues and use similar solutions. In fact, Sidney's design requires only that there is some low-level mechanism that allows multiple modifications made to volatile data to be transferred to disk atomically. The design issues specific to Sidney are concerned with how to use this low-level mechanism to provide a transactional heap. I will focus primarily on issues that arise because of caching persistent data in volatile memory.

The section begins by describing Sidney's basic architecture, in particular the representation of the heap, both on disk and in volatile memory. It then considers how modifications to the volatile cache are tracked and transferred to disk. It then discusses the performance ramifications of the design decisions. Details of the implementation of this design are found in Chapter 5.

2.2.1 Transaction Architecture

Figure 2.1 shows the basic block diagram of the transaction design. Since persistent data must survive system crashes, the persistent heap must maintain a representation on stable storage, shown here as the stable image. To achieve good performance, the persistent heap must also maintain a representation in volatile memory, shown here as the volatile image.

The diagram shows that only `commit` accesses the stable image; all the other operations act only on the volatile image. The addition of transactions does not change `read`. Changes to `alloc` are discussed in Chapter 4. `Commit` uses the Transaction Manager (TM) to transfer changes in the volatile image to the stable image. So that the Transaction Manager can find the modified locations in the volatile image, `write` records the location of any modifications to the volatile image in the write log.

Given two images, a key design issue is how to map data between them. Many systems attempt to keep only a portion of the persistent data cached in volatile memory [5, 39]. Such systems allow manipulation of persistent heaps that are much larger than the physical or even virtual address space on the machines available. However, it complicates the process of bringing data from stable storage into volatile memory. The design and implementation of such systems is an active area of research [39, 58, 60].

In the interest of simplicity, Sidney does not attempt to map selected portions of the persistent heap into volatile memory. Instead, in the absence of uncommitted `writes`, the stable image is an exact copy of the volatile image and `commit` must maintain this

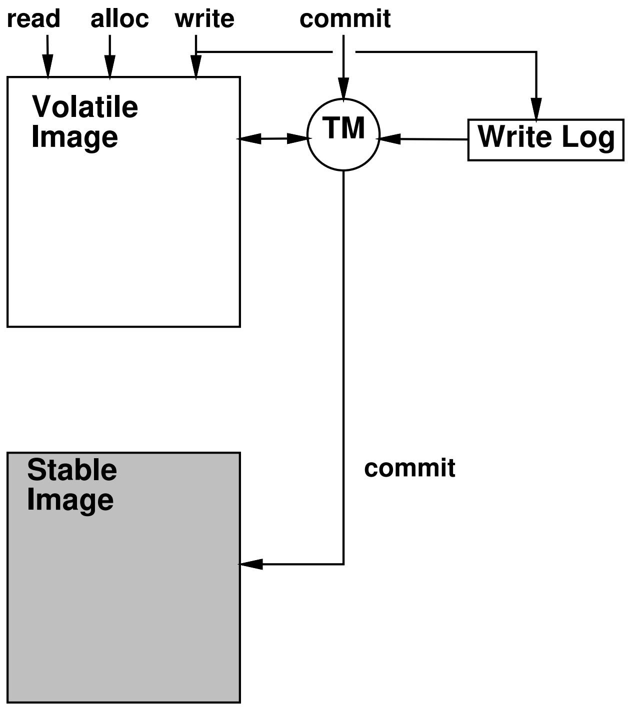


Figure 2.1: Transaction Architecture

correspondence. When the system is started, the stable image is copied into volatile memory at a fixed location, thus maintaining the validity of any pointers in the heap. If it is impossible to map the heap into the same location, the system can relocate the heap in volatile memory and then write the new image to stable storage. Otherwise no translation is done between the two images.

The decision to map the stable image directly into volatile memory limits the size of the persistent heap to the size of virtual address space of the machine in use and pragmatically to the size of the available physical memory. Physical memories with many tens of megabytes are now commonplace, and those with hundreds of megabytes are not uncommon. Persistent heaps with memories of these sizes can accommodate many useful applications, so this size restriction is not a fundamental limitation. Furthermore, I expect many of the techniques developed for Sidney, in particular those for concurrent garbage collection, will complement those that can be used to accommodate larger heaps. Many researchers [39, 60] are working on how to allow heaps larger than physical memory; I discuss some of the work in Chapter 11, the related works. Finally, if it proves impossible to achieve good performance even in Sidney's more limited domain, it seems unlikely that the more complicated domain will offer acceptable performance. Thus, the current work is a key milestone to establishing the general performance characteristics of general-purpose persistence.

2.2.2 Tracking Modifications to Persistent Data

When the client writes into the volatile image, the volatile image becomes inconsistent with the stable image. `Commit` must make the stable image consistent with the volatile image. Consistency can be achieved by either atomically replacing the entire stable image with the volatile one, or by atomically updating only the portions of the heap that have changed. The desire to support small fine-grained transactions dictates the use of the latter approach.

Sidney tracks the location of `writes` to the volatile image implicitly. Implicit tracking of `writes` is done using a class of techniques called *write barriers*. Write barriers are also needed in other contexts such as generational garbage collectors and distributed shared memory systems [12, 56]. Techniques for implementing write barriers have been carefully studied by Hosking [24].

Sidney uses a simple write barrier in which the compiler emits code to record the location of each `write` in a log. In practice, in addition to the location of the `write`, the value overwritten is also recorded, allowing the write log to serve as an undo log also. Logging undo information supports both transaction `abort`, which is not a concern in this dissertation, and certain aspects of garbage collection discussed in Chapter 5.

2.2.3 Atomic Update of the Stable Image

`Commit` must make the stable image consistent with the volatile image. As already mentioned, the design assumes the existence of a low-level mechanism for updating the disk atomically. This assumption coupled with the fact that all modified locations can be found in the log makes the implementation of `commit` straightforward. When a `commit` is performed, the Transaction Manager first begins a low-level transaction. The Transaction Manager then traverses the write log and records the new values of the modified locations with the low-level mechanism. Finally, the Transaction Manager ends the low-level transaction, which forces the new values to disk, completing the atomic update of the disk.

2.2.4 Performance Ramifications

Mapping the entire persistent heap into memory incurs a cost at system startup proportional to the size of the heap. Since persistent data are directly accessible by the client, the cost of a `read` is the same as for accessing non-persistent or *transitory* data.

Each `write` has some small additional cost due to logging. Due to the effects of the memory hierarchy, this cost is not constant, but it can be bounded by a constant. Thus the total cost of logging `writes` is O(# of `writes`).

Each `commit` has a fixed cost plus a cost for each `write` done during the current transaction. `Commit` also has the cost of atomically updating the stable image, which is some fixed cost plus some overhead for every byte that must be transferred to disk. The number of bytes transferred to the disk is some fixed amount plus some amount for each location modified. The number of locations modified cannot exceed the number of `writes`. Thus, asymptotically, the cost of all `commits` is O(# of `writes` + # of `commits`).

From this analysis we can conclude that the cost of transactions is O(# of `writes` +

of `commits`). Any other solution that tracks `writes` will also be O(# of `writes` + # of `commits`) as argued in Section 2.3. A detailed examination of the constant factors involved is a central part of the performance evaluation beginning in Chapter 6.

Despite the equivalence with respect to asymptotic complexity, intuitively, logging every `write` seems expensive. However, this decision was partly influenced by the programming language involved, SML. In SML many operations are done by binding a new value, rather than overwriting an old one as would be done in languages like C. Thus, `writes` are less frequent. Furthermore SML programmers generally attempt to minimize the use of `write` in other circumstances as well, which further reduces the cost of logging. In fact, to support its generational garbage collector, SML/NJ already logs all `writes` to values that may contain pointers.

2.3 Design Alternatives

In this section I consider some of the possible design alternatives. I will argue that either they do not satisfy the system requirements, or at the least they do not offer improved asymptotic performance.

2.3.1 Explicit Logging of `Writes`

An obvious modification of the design is to require that the programmer explicitly log the locations of all modifications. In fact, explicit logging [35] is exactly the interface provided by RVM. The principal disadvantage of explicit logging is that the programmer might not log a modified location. This error would cause an update to the persistent heap to be lost and allow data to become inconsistent. The possibility of inconsistent data is fundamentally at odds with Sidney's safety goals.

An apparent advantage to the use of explicit logging is increased performance. If the programmer happens to know that some location or range of locations (such as an array) will be updated multiple times, then that fact may be recorded only once. However, while this may result in savings in some cases, it still has the same asymptotic complexity as the implicit approach in the worst case, since it is possible that all `writes` are made to disjoint noncontiguous locations. It is also possible for explicit logging to cause more logging as well. For example, suppose that a function logs changes to a data structure passed in as an argument. If the programmer does not realize this, the programmer may also log the change, since that is the safest thing to do. Implicit logging only logs the `write` when it occurs, so there is no double logging. In inspecting code that uses explicit logging, I have found examples of both avoiding logging multiple `writes` to structures like arrays and of double logging, so both of these possibilities occur in practice.

2.3.2 Using Virtual Memory to Log `Writes`

Virtual memory can also be used to implement the write barrier. The basic technique is as follows. Initially pages are write-protected. When the user program `writes` to a location a fault occurs. The system traps this fault and logs that the entire page may have changed. It then un-protects the page and resumes the program. `Commit` must then force every page recorded in the log to disk and then re-protect the pages.

There are several disadvantages to this approach. It has an asymptotic complexity of O(min(# `writes`, # pages in address space)). Since there is no fixed size of the address space (except for the size of an address itself), this technique is effectively the same as the other approaches. For small-grained transactions this technique forces an entire page to be written to stable storage when writing only a few bytes would suffice. Or if a pre-modification version of the page is retained, only the differences need to be written out, but at the least the whole page will have to be examined. Any such implementation will also have limited portability because of operating system dependencies. Finally, using this kind of write barrier makes it difficult to distinguish `writes` from different threads, thereby making multi-threaded transactions harder to implement.

2.3.3 Checkpointing

All of the approaches involving write barriers, implicit or explicit, are designed to allow only the modified data to be transferred to disk. Another approach would be simply to write the entire persistent heap to disk upon a `commit`. A principal advantage of this approach is that it is simple to implement. Obviously, this approach has poor performance when only a few locations are changed by a transaction. The technique of attempting to checkpoint only the modified portions of the heap is essentially the same as the technique used by Sidney or other techniques discussed above. Checkpointing also makes multi-threaded transactions harder to implement, because at any given time some transaction may not be prepared to `commit`.

2.3.4 Omitting Transactions

Systems [5] have been built that support persistence but not transactions. Such systems still cache persistent data in volatile memory and thus must face the problem of when to update the stable version. The typical approach is to update the stable version only at the end of a session. In essence this model is a transaction-based system in which committing also shuts the system down. Sidney's design is more general than this.

Chapter 3

Garbage Collection

Dynamic allocation of persistent data is one of the basic services provided by Sidney. Static allocation of storage is simply not flexible enough to support general-purpose computing, as evidenced by the support for dynamic allocation in almost all programming languages. The long-lived nature of persistent data makes it even less likely that any pre-arranged static allocation will be suitable over the long run; dynamic allocation is therefore especially important for persistent storage.

If storage can be dynamically allocated, it must also be dynamically deallocated. This requirement is especially true of long-lived servers, which are important clients of transactional storage. An obvious solution is to require that the programmer explicitly deallocate storage that is not in use. Explicit deallocation has proven to be difficult to do correctly and to be a source of errors that are long-lived, hard-to-detect, and hard-to-correct [49].

Instead, Sidney uses implicit storage reclamation, or *garbage collection*. Garbage collection is free from the kind of safety concerns that plague explicit storage management. The potential problem with garbage collection is poor performance. Although the notion that garbage collectors are slow compared to explicit storage managers is common, there is good evidence that this is not the case. For example, Zorn [63] shows that the throughput of a simple garbage collector is comparable to that of a number of explicit storage managers for transitory data. In Part II of the dissertation, I show that the throughput achieved by Sidney is comparable to that achieved by an explicit persistent storage manager.

The other performance problem associated with garbage collection is the *garbage collector pause*. Garbage collectors must occasionally pause their client program, but if these pauses are long they will be disruptive. An important contribution of Sidney is a new garbage collection technique, *replicating collection*. This technique allows the garbage collector to execute concurrently with the client, thus minimizing the duration of the garbage collector pauses. Replicating collection is especially well suited to collecting persistent heaps.

This chapter begins by discussing how the garbage collector design meets both the safety and performance goals of Sidney. The bulk of the chapter focuses on the design of the garbage collector. This section begins with a review of copying garbage collection. It then discusses replicating collection in general and how it is applied to collecting a transactional heap. The chapter concludes by offering an evaluation of some of the design alternatives.

3.1 Safety and Performance

If the programmer is required to free unused storage explicitly, then it is possible that storage that should be freed will not be. This situation can lead to the amount of storage needed by a program growing without bound. If a program runs for a long time, as do many servers, this situation is very likely to arise. Persistence makes this problem even worse: storage that is lost in a persistent heap can never be reclaimed, except when there is some special-purpose way to recreate the heap from scratch.

If the programmer is required to free unused storage explicitly, then it is also possible that storage that should not be freed will be. Once an object has been freed incorrectly, then the storage associated with the object will be reallocated to a new object. Should this error occur, there would be two objects using the same underlying storage; such an error compromises the integrity of both objects. If these objects are persistent, then the error is even worse: persistent objects contain valuable data; otherwise they would not be persistent. Furthermore, since persistent heaps may be used over a long period of time by more than one program, finding the source of such error becomes especially challenging.

To avoid the safety problems of explicit deallocation, Sidney uses implicit storage reclamation, or *garbage collection*. GC does not allow storage that is in use to be reclaimed and it limits the time during which storage that is not in use is allowed to go without being reclaimed. These properties are crucial to Sidney's basic goal of safe storage management.

The challenge that garbage collection presents is how to achieve good performance. Two kinds of performance difficulties are associated with GC: high CPU overhead and therefore poor throughput, and long GC pauses and therefore poor latency. Sidney attacks these problems by using a new garbage collection technique that allows the collector to execute concurrently with the client. This approach can both improve collector throughput and alleviate the problem of GC pauses.

This new concurrent garbage collection technique, *replicating collection,* minimizes the interactions that the collector has with the client. This technique also minimizes interactions with the transaction manager, making replicating collection ideally suited for collecting transactional heaps. Furthermore, the main overhead imposed by replicating collection is a write barrier that is also needed to support transactions, making replicating collection ideal for collecting transactional heaps.

3.2 Copying Garbage Collection

Copying collection is a broadly used and well studied technique; this discussion is only meant to touch briefly on the key ideas and terminology. For more details, a survey may be found in Wilson [59].

Copying garbage collection, which is Sidney's basic garbage collection technique, is an example of a larger class of techniques called *tracing garbage collection.* Tracing collectors work by finding all of the storage a program may legally access; this storage is *live* and must be retained. Any storage that cannot be accessed is *garbage* and should be reclaimed. Tracing collectors find all accessible storage by starting from a set of locations, called the

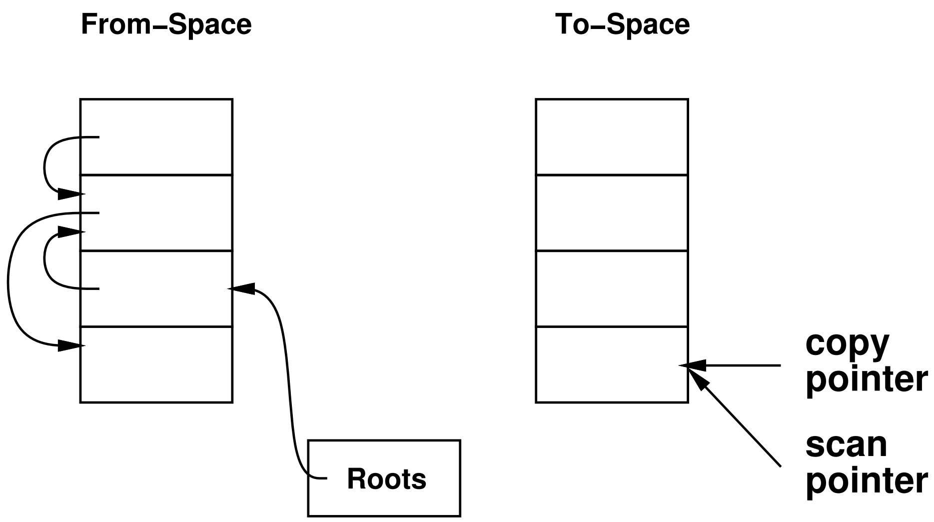


Figure 3.1: Beginning of Copying Collection

roots, that are known to be live. The roots typically include the registers, the stack, and some global data. Many objects in the heap contain pointers that refer to other objects. Any object that can be accessed by dereferencing pointers starting from the roots is live. The collector finds all the live (i.e., accessible) objects by starting from the roots and tracing pointers until all reachable objects have been found.

Copying collection differs from other tracing collectors in what happens to the objects when they are found to be accessible. Accessible objects are copied from their current location in *from-space* to *to-space*. A *forwarding pointer* to the new version is left behind, so that references to the old version can be redirected.

When the client runs out of space, the collector stops the client, copies all the accessible objects to an empty region of memory, reclaims the old storage, and restarts the client. Since the collector stops the client during the copy, the basic technique is known as *stop-and-copy* collection. The length of the pause is proportional to the amount of data copied. I refer to this amount as the *livesize*.

Figure 3.1 shows the heap when the copying collection process begins. Each object is represented by a box and pointers are represented by arrows. Initially all objects are in from-space. In this picture, the second object from the bottom is directly accessible from one of the roots and the bottom three objects are live. The top object cannot be reached by traversing pointers from the root, and thus is garbage. To-space is where the objects will be copied and thus is empty at the beginning of collection.

Figure 3.2 shows the collection in progress. The `copy` operation is one of the basic operations used by copying collection. In the figure, the first object reachable from the root has been `copied`. This has caused it to be copied to an empty part of to-space, at the location referred to by the *copy pointer*. `Copy` leaves behind a *forwarding pointer*, shown in the figure as a dashed arrow, that allows the to-space version of an object to be found. `Copy` also causes the copy pointer to advance to the next free to-space location. `Copy` does not update the pointer contained in the object; this pointer still refers to an object in from-space.

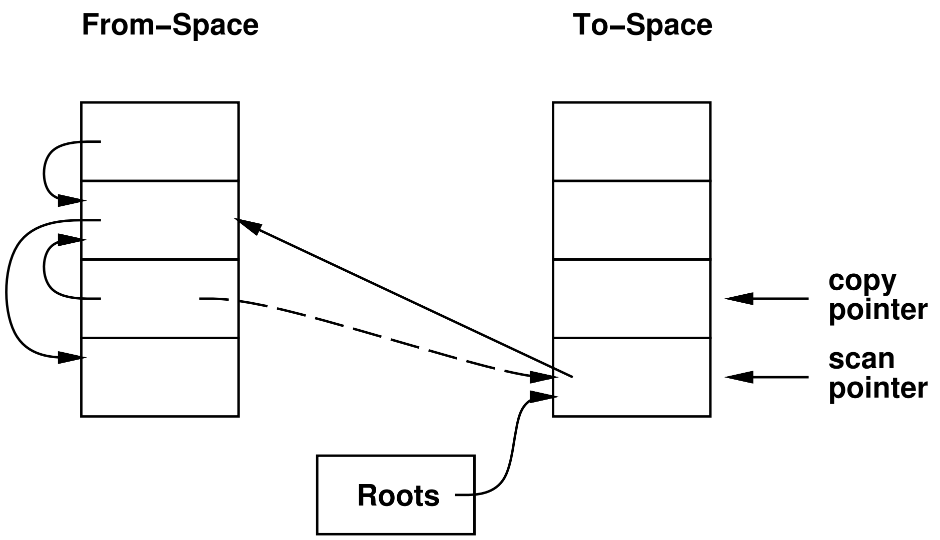


Figure 3.2: Copying Collection in Progress

`Scan` is the other key operation in copying collection. In the figure, the root has been `scanned`. `Scan` updates any pointers contained in an object to point to to-space versions of the objects to which they refer. If the object pointed to by the object being `scanned` has not been `copied` and thus does not have a to-space version, then `scan` forces it to be `copied` so that it will. Before the collection completes, each live object will first be `copied` and then `scanned`.

When collection starts, both the copy and scan pointers point to the beginning of to-space. The first step of a collection is to `scan` the roots, which causes the copy pointer to advance but does not change the scan pointer. The algorithm then proceeds to `scan` the object pointed to by the scan pointer. This `scan` forces any uncopied objects referred to by the object being scanned to be copied, and all the pointers in the scanned object to be updated to point into to-space. The scan pointer is then advanced to the next unscanned object. An important invariant is that all objects below the scanned pointer have been scanned and contain only to-space pointers.

The process terminates when the scan pointer reaches the copy pointer, as shown in Figure 3.3. At this point all reachable objects have been copied and scanned. The collector renames to-space to be from-space; this action is referred to as the *flip*. Finally the client resumes execution using the new heap.

3.2.1 Performance Implications

Each garbage collection will take time proportional to the amount of live data that must be copied. Since the collector stops the client while collecting, this implies that the duration of a GC pause depends linearly on the livesize. Large heaps lead to unacceptably long pauses. An important goal of Sidney is to limit the length of these pauses.

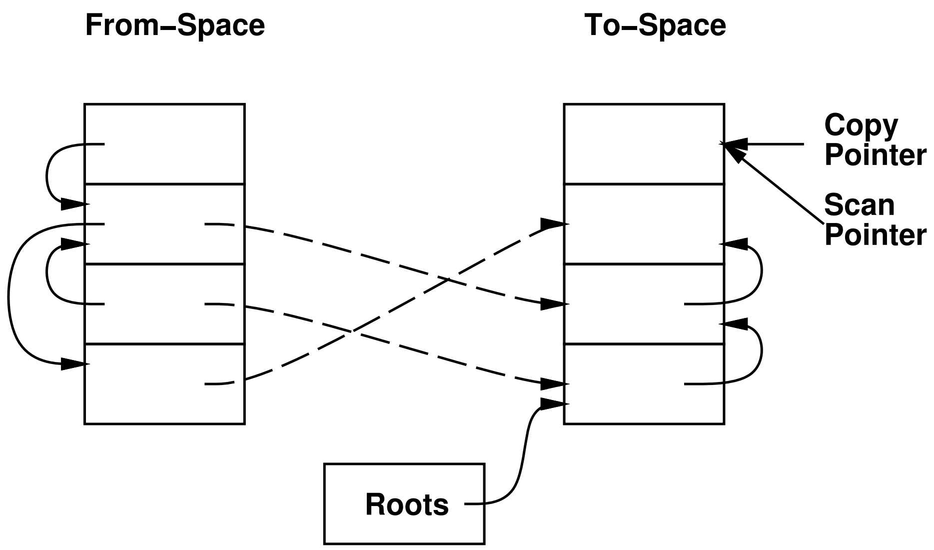


Figure 3.3: Completed Copying Collection

Besides reclaiming storage, copying collection has another benefit: It compacts the live data into a contiguous region. Compaction has been shown to result in improved virtual memory behavior [61]. Rearranging and compacting objects may prove even more useful in the context of persistence, because using such rearrangements may help optimize disk accesses. Demonstrating this potential advantage is beyond the scope of this dissertation.

Since large blocks of contiguous storage are freed at once, copying collection allows allocation to be performed cheaply. The allocator maintains a *free pointer* into a contiguous region of memory. Allocation is a matter of testing a limit value and incrementing the free pointer. Often these limit checks can be amortized over several allocations and allocation is made an in-lined operation. For SML/NJ an allocation takes (on average) just a fraction more that one machine instruction.

3.2.2 Generational Collection

Generational collection is an important enhancement to the basic technique of tracing collection. Generational collection depends on the following empirical observation: recently allocated or *young* objects are more likely to become garbage than less recently allocated or *old* objects [33, 56]. Although not guaranteed to hold, this property has been observed in many systems and applications [4, 33, 56]. Intuitively, young objects often contain temporary values and thus die young, while older objects are often part of permanent data structures and thus are likely to remain live.

The cost of copying collection depends on the amount of live data copied. Generational collectors reduce the cost of each collection by keeping young objects separate from older objects, and collecting the regions containing young objects more frequently that those containing old objects. These regions are known as *generations*. This technique results in a savings because the young objects are less likely to be live, and thus the collections that focus on the young objects will have fewer objects to copy. Since the collections that focus on the young objects have relatively few objects to copy, pauses caused by these collections are brief as well.

Introducing generations has an important consequence. All pointers from an older generation into a younger one must be used as roots during the collection of the younger generation. An obvious way to find such pointers is just to examine all the objects in the older generation. Examining all older objects is simple, but has a cost proportional to the number of objects in the older generations. A more common way of finding cross generational pointers is to note that such pointers can be created only by `writes`. This observation suggests using a write barrier similar to the one discussed in Section 2.2.2 to track cross-generation pointers. For generational collection, only `writes` that create pointers from older generations to newer ones must be tracked, whereas for transactions all `writes` into the volatile image must be tracked.

3.3 Concurrent Replicating Collection

The basic problem with stop-and-copy collection is that the client is stopped during collections, creating a pause. Since collections take time proportional to the livesize and the livesize can be large, garbage collector pauses can become disruptive. Concurrent garbage collectors permit the client to execute while the garbage collection is in progress, and thus limit the duration of garbage collection pauses. Concurrent collection also offers the possibility of increased performance through the use of parallelism. Parallelism arises most obviously on machines with more than one processor, in which case the collector can run in parallel with the client. In the context of persistence, however, another important source of parallelism arises: overlapping computation with I/O. In a system using persistence, both the client and the storage management system will do I/O; the ability to overlap computation with I/O will prove to be significant, as we will see in Part II of the dissertation.

Concurrent collection allows the operations of the client and the collector to be interleaved in any order, yet the effects of the garbage collector must not be observable by the client. In previous concurrent copying garbage collection designs [3, 6], the interactions between the client and the collector lead to complex and expensive synchronization requirements. Sidney uses a new technique called *replicating garbage collection* that requires that the collector replicate live objects without modifying the original objects. Since the original objects are not modified, interactions with the client are minimized, making this design attractive for use in a concurrent collector. The details of how replicating collection works is the focus of the rest of this section.

3.3.1 The Client Uses From-Space

When copying an object, the usual technique employed by stop-and-copy collectors destroys the original object by overwriting it with a forwarding pointer. Therefore, concurrent collectors using this technique must ensure that the client always accesses only the relocated (to-space) copy of an object. This requirement is referred to as the *to-space invariant*.

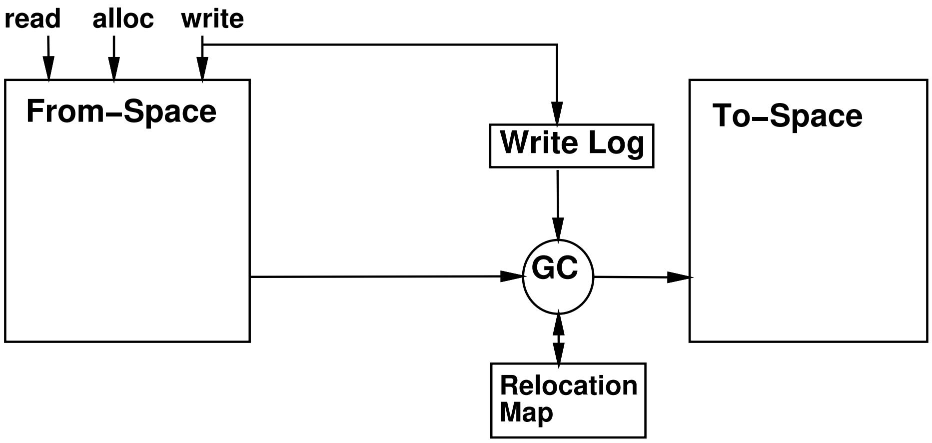


Figure 3.4: Replicating Collection

Replicating collection differs from previous concurrent copying collectors because copying is performed non-destructively. Non-destructive copying means that the to-space invariant is no longer required. Conceptually, whenever the collector copies an object it stores a relocation record in a relocation map, as shown in Figure 3.4. Chapter 5 gives details about how the relocation map is implemented. In general, when using replicating collection, the client may access the original object or the relocated objects [42], and is oblivious to the existence or contents of the relocation map. In Sidney, the client always accesses only the original (from-space) object; this requirement is referred to as the *from-space invariant*.

3.3.2 `Writes` are Logged

After the collector has replicated an object, the original object may be modified by the client. In this case, the same modification must also be made to the replica before the client can safely use the replica. Replicating collection uses a write barrier to record all `writes` in a write log, as shown in Figure 3.4. The collector uses this log to guarantee that the to-space replicas are up-to-date before the algorithm terminates. The collector reads the log entries and applies the `writes` to the replicas, a process referred to as *write forwarding*.

3.3.3 Collector Execution

The collector begins execution when some predetermined condition is triggered by the client. Typically this would be when the free pointer reaches some limit. The collector executes concurrently with the client and replicates all the objects that are accessible to the client. The collector creates replicas of the objects pointed to by the client's roots, and then scans the objects in to-space. It also reads the write log and forwards `writes` to the appropriate to-space replicas, re-scanning these replicas if needed.

One subtle point is that initially, the collector cannot `scan` the roots or other data-structures to which the client has access. Scanning them would cause the root to point into to-space; this would allow the client to access to-space objects, violating the from-space invariant. Instead, Sidney `copies` the locations referenced by the roots, but does not update the roots themselves. The roots are not updated until during the flip, when all references to from-space must be updated to refer to to-space atomically.

A collection is complete when the write log is empty, the client roots have been copied, and all of the objects in to-space have been scanned. This is called the *flip condition.* When the flip condition is met, all objects reachable from the roots have been replicated and are up-to-date. The collector can now flip by using the following steps: the collector halts the client, reestablishes the flip condition, updates the client's roots to point to the corresponding to-space replicas, discards the from-space, renames to-space to be from-space, and finally, resumes the client. The collector must reestablish the flip condition because the roots may have changed and there may be new entries on the write log. Meeting this requirement may cause additional scanning, copying, and write forwarding. The collector monitors this work, and if it exceeds some parameterizable threshold, returns control to the client without flipping. This technique limits the amount of work done while the client is paused.

3.3.4 Client Interactions

Although the garbage collector executes concurrently with the client, the from-space invariant ensures that there is no low-level interaction between the collector and client. The client executes machine instructions that `read` and `write` the objects that reside in from-space. The collector reads the objects in from-space and writes the objects in to-space. Conceptually, the relocation map shown in Figure 3.4 is used only by the collector.

The collector does interact with the client via the write log and the client's roots. The collector must occasionally obtain an up-to-date copy of the client's roots in order to continue building to-space replicas based on the current roots. Also, the collector reads the write log, which is being written by the client. These interactions do not require that the client be halted, but in practice, Sidney interrupts the client briefly when these interactions are required.

When the collector has established the flip condition, it must halt the client in order to atomically reestablish the flip condition and update the client's roots. After the roots have been updated, the client can resume execution. The duration of this pause in the client's execution depends on the synchronization delay due to interacting with the client thread, the amount of work to reestablish the flip condition, and the size of the root set. This is the only action required to synchronize with the client. The limited client–collector interaction makes this design easier to implement than designs in which the need to synchronize is more pervasive [6, 21]. It probably is cheaper as well.

3.4 Transactional Heap Collection

Replicating collection is ideal for persistent heaps that support transactions. Replicating collection has no low-level interaction with the client, thereby also limiting the interaction between the collector and transaction system. Moreover, the main source of overhead for replicating collection, the write barrier, is already needed to support transactions.

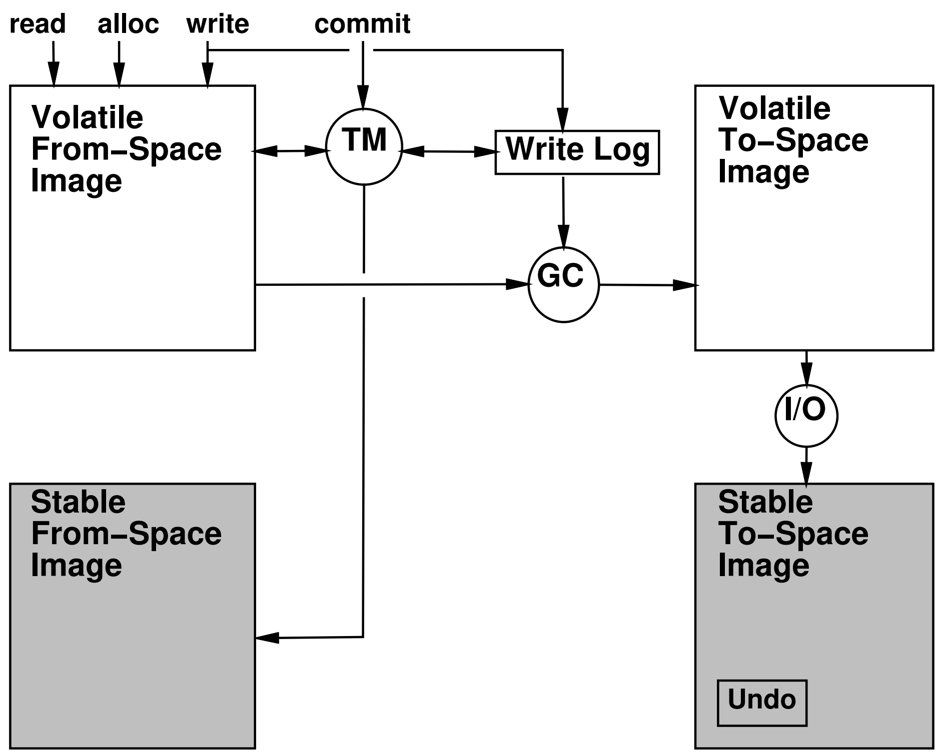


Figure 3.5: Transactional Heap Collection

Figure 3.5 shows a basic block diagram of Sidney's garbage collector design. Since accessing stable storage is expensive, the collector uses the volatile image as from-space. The figure also shows that the to-space used by the collector is composed of both a volatile and a stable image. Splitting the to-space means that the volatile to-space must be transferred to disk before the collection can complete. The I/O thread shown in the diagram does this transfer. The transfer of the volatile to-space can proceed concurrently with copying data from the volatile from-space to the volatile to-space.

Since the from-space invariant means that the client is unaware of the collector, the Transaction Manager continues to operate in exactly the same manner as described in Chapter 2. The write log is used by both the collector and the transaction manager to track `writes` into the volatile from-space. The garbage collector uses replicating collection, as described above, although before a flip, the volatile to-space must have been written to disk. When the flip occurs, the volatile to-space becomes the volatile from-space and the stable to-space becomes the stable from-space. If a system failure occurs, the stable from-space will be used to restart the system. This requirement means that the flip must be done atomically with respect to failures.

An important consequence of using the volatile from-space as the basis of GC is that any uncommitted data in from-space will be copied to volatile to-space by the collector, and from there into the stable to-space by the I/O thread. In the case of failure, on recovery only committed data must be provided to the client. This requirement means that the uncommitted data must be dealt with.

Sidney uses a simple approach to deal with uncommitted data in the stable to-space. Recall that to support transactions fully, including `abort`, the write log contains the old values of the overwritten data and so can be regarded as an undo log as well. The log is included in the data written to stable to-space. If the system fails and recovery is needed, the log can be used to restore the committed values, just as if the transaction that was on-going during the flip had been aborted. Any objects that happen to be included in the stable to-space only as a part of the undo log will be garbage collected during the next persistent heap collection.

3.5 Design Alternatives

In this section, I consider alternatives to the use of concurrent replicating collection. In most cases these alternatives do not meet either Sidney's safety or performance goals. At least one alternative, mark-and-sweep collection, could meet these goals, but is incompatible with the design for supporting orthogonal persistence (see Chapter 4).

3.5.1 Explicit Deallocation

The most obvious alternative to garbage collection is to force the programmer to deallocate storage explicitly. A potential advantage of explicit deallocation is better performance, but the use of explicit allocation is counter to the safety goals of Sidney.

Since performance is the primary motivation for using explicit deallocation, one would hope that it offers some inherent performance advantage. This is not the case. One way explicit deallocation might show improved performance would be if it guaranteed that there would never be an unbounded pause during allocation or deallocation. No known algorithm offers such a guarantee, although it is true that long pauses caused by explicit allocation-deallocation are typically much less frequent than those caused by GC.

Another way that explicit deallocation could offer better performance would be if it had inherently better throughput. One disadvantage of using explicit management is that allocation must use a free list rather than just increment a pointer. Using a free list makes allocation much more expensive. Freeing unused objects is the other basic cost of explicit deallocation. The cost of copying collection depends on the number of live objects. These two factors, the number of live objects and the number of frees, are incomparable, which makes it impossible to compare the two techniques asymptotically. The best one can argue is that explicit storage management favors the case where there are (relatively) fewer allocations and where these allocations are mostly of long-lived data. GC tends to make allocation cheap and favors the case where objects are transient.

Zorn [63] has compared the performance of a conservative mark-and-sweep collector to that of explicit storage management for transitory (i.e., not persistent) data. His results show that garbage collection can be competitive with explicit deallocation. In the performance section, I compare the cost of Sidney to that of an explicit allocator for persistent storage. This comparison shows that the overhead of GC is competitive with the overhead of explicit deallocation.

Another disadvantage of explicit storage management is that it does not compact the heap. Not only does this reduce locality, but allocation may fail due to fragmentation even when there is sufficient free storage in the heap. A common approach used to deal with the problem of fragmentation is to attempt to coalesce contiguous portions of free data. Unfortunately, while coalescing decreases the problem of fragmentation, it also makes long pauses much more frequent.

3.5.2 Reference Counting Collection

Reference counting is an implicit storage management that requires that each object track the number of references to itself and free itself when no references are left. There are two basic problems with this approach. First, it requires work on each creation and deletion of a reference, which are very common operations. This means that making reference counting efficient is quite hard [14, 16]. The more serious problem is that reference counting cannot collect circular structures, which leads to the possibility of garbage accumulating. It also shares many of the problems of explicit storage management, such as the potential for fragmentation and the need to allocate from a free list.

3.5.3 Mark-and-Sweep Collection

Mark-and-sweep collectors are the other examples of tracing collectors. When a live object is found, it is marked as used. Once all live objects have been found, the garbage objects are located by sweeping memory and adding them to a free list.

Whether mark-and-sweep or copying collection is to be preferred in general is an open research question. A current trend is to build hybrid collectors that combine aspects of both. The problems with mark-and-sweep collectors are much the same as with other techniques that do not relocate objects: allocation from free lists is (relatively) expensive, the heap is not compacted, and fragmentation may become a problem.

A final issue with mark-and-sweep collection is its asymptotic complexity. Since the sweep phase must examine the entire heap for unmarked data, the cost of mark-and-sweep is O(size of heap) rather than O(livesize), as it is for copying collection. In practice, however, the marking phase dominates the cost of the sweeping phase. In addition, the sweeping phase can be piggy-backed on allocation, essentially making allocation more expensive, but also making it less likely that sweeping will be a bottleneck [8].

Despite these issues, mark-and-sweep collection is still a reasonable candidate for concurrent persistent heap collection. In fact, as discussed in Chapter 11, the only other implementation of a concurrent persistent (but not transactional) heap collector used mark-and-sweep collection. Sidney uses copying collection for two simple reasons. First, some of the features I wished to explore were incompatible with mark-and-sweep collection; I discuss this issue more fully in Chapter 4. The other reason was purely pragmatic — SML/NJ already uses a copying collector and changing it to be mark-and-sweep would seem to serve no useful purpose.

3.5.4 Generational Collection

When generational collection is used, most collections are done on only the young objects. Since these objects are likely to die, the collections focusing on them are short. However, this does not make generational collection a general solution for the problem of collector pauses. There is still a need to occasionally collect all the objects, not just the young ones. This need means that the longest pauses are much less frequent, but some form of concurrent collection is needed to avoid long pauses altogether.

Hudson and Moss [28] have proposed a new technique for incrementally collecting the older generations. Recently, the first implementation of this approach has been done for Beta, by Seligmann and Grarup [52]. The results are good and this technique may be able to avoid need for long pauses without using concurrency. No one has yet tried to discover if Hudson and Moss's algorithm is compatible with persistence and transactions, but this would certainly be worth consideration.

3.5.5 To-Space Invariant Concurrent Copying Collection

Section 3.3.1 mentions that other concurrent copying GC algorithms exist that use a to-space invariant. Two designs for transactional heap collectors have been based on these techniques, although none has been successfully implemented. Since these designs are the ones most directly related to mine, I will discuss them in some detail in Chapter 11 on related work.

Chapter 4

Orthogonal Persistence

If performance and cost were not an issue all data could be made persistent. In practice, however, persistent data are more expensive to manage than transitory data in terms of time and other resources, such as disk space. Thus, it is important that any system that supports persistent data also support data that are not persistent or *transitory*.

Drawing a distinction between persistent and transitory data means that there must be some way to identify which objects should be persistent. There are two basic approaches, *explicit* and *implicit*. In an explicit system, the programmer must explicitly identify what data should persist. The simplest way to do this is to have a special version of the allocator that allocates only persistent data. Another common approach is to associate persistence with the type or class of the data; this allows the language system to tell when the persistent allocator must be called. These approaches are discussed in Section 4.3.

Like garbage collection, the implicit approach, *orthogonal persistence* [5], is safer and more convenient for the programmer. The basic idea is simple. In the persistent heap there is a special location called the *persistent root*. Any data that are reachable from the persistent root by dereferencing pointers are persistent. Thus, if a `write` makes a transitory object reachable from the persistent root, the transitory object becomes persistent. We will see that it is the responsibility of `commit` to actually promote the transitory object into the persistent heap. The safety advantage of this approach is that on recovery there is no possibility that a persistent object references a transitory object that no longer exists.

In the sections that follow, I first discuss how the design supports orthogonal persistence. Next, I discuss a subtle performance problem with the design and show how it can be corrected using techniques based on replicating collection. Finally, I discuss alternatives to orthogonal persistence.

4.1 Safety and Performance

Providing support for transitory data is primarily motivated by concerns for cost and performance. Persistent data must be maintained both on disk and in memory, which is inherently more expensive that just maintaining the data in volatile memory. In particular, Sidney's implementation may allocate as much as ten megabytes of data per second. To

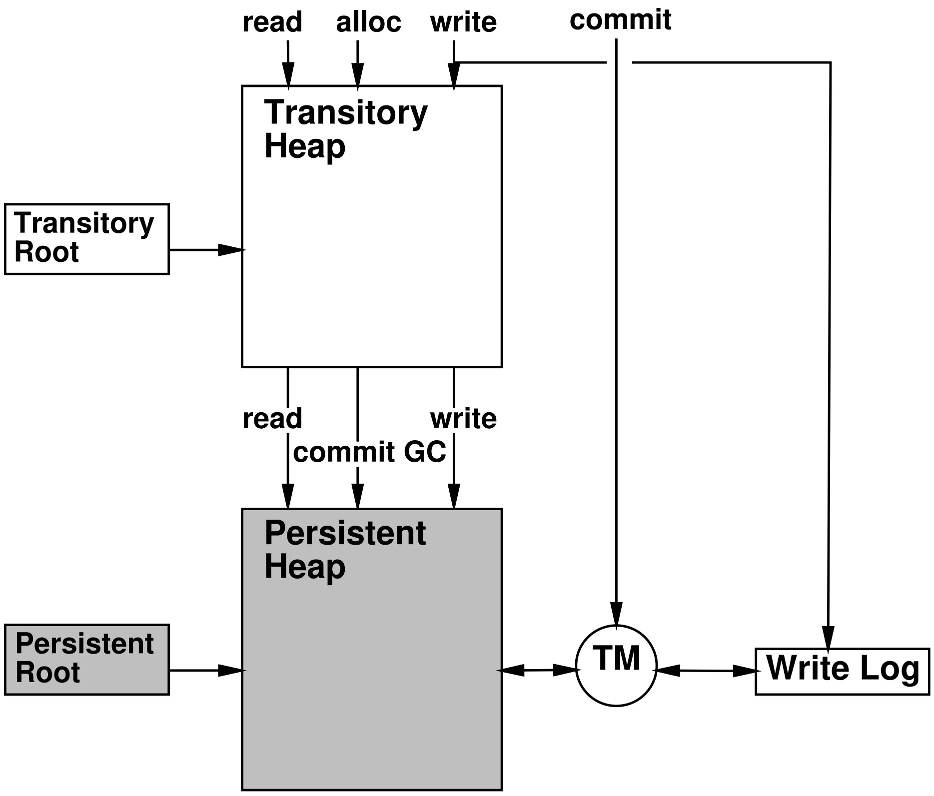


Figure 4.1: Adding a Transitory Heap

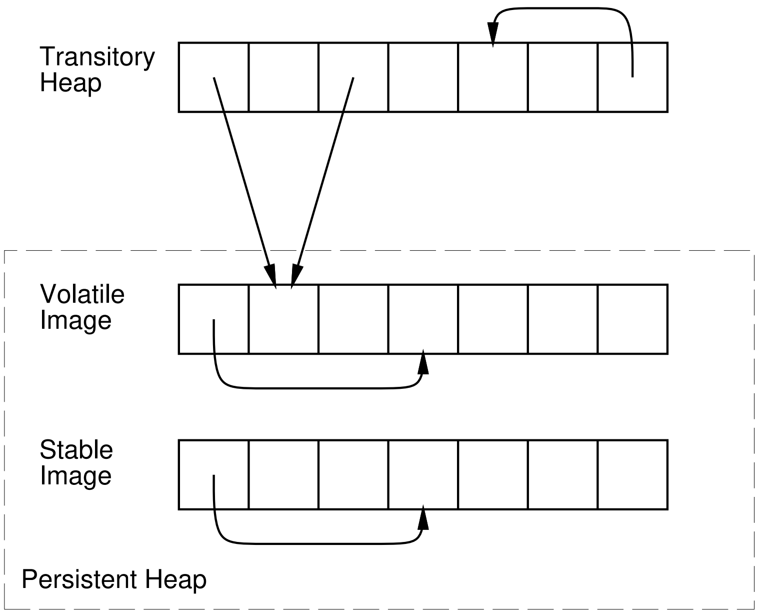


Figure 4.2: A Committed State

transfer data to disk at this rate would be a significant challenge. Thus, Sidney provides support for transitory data.

The choice to use orthogonal persistence is motivated by both safety and issues of programmer convenience. The safety issue is simple: in a basic explicit persistence system, it is quite possible for the programmer to create persistent objects that refer to transitory ones. If the system crashes, then on recovery these transitory objects will no longer exist and the references will be invalid. Of course, it would be possible to avoid this problem by not allowing persistent objects to reference transitory objects, but that is very restrictive and is also likely to be expensive to enforce. Worse and harder to avoid, is the possibility that the programmer will neglect to make some critical data persistent. The programmer convenience issue is also simple. In a system that supports orthogonal persistence, the programmer does not have to worry about whether an object should be made persistent or not; the system figures it out. If the programmer wants to make sure a particular object is persistent, that can be arranged by making that object reachable from the persistent root. The programmer does not need to worry about the objects referenced by the newly persistent object, as the system handles that automatically.

The question is how to implement orthogonal persistence so that it has good performance. This implementation is easier in a garbage-collected system, since the garbage collector can already perform the kind of reachability analysis that orthogonal persistence requires.

As we explore the design, we will find that a subtle technical point makes it appear that any `commit` will need to examine all of the data in the transitory heap. This requirement is antithetical to the goal of having small transactions be fast, since even a transaction that changes a single byte will need to examine the transitory heap, which may be very large. Fortunately, it will turn out that replicating collection will enable a clever optimization that avoids the need to examine the entire transitory heap, and thus will allow an implementation of orthogonal persistence that has acceptable asymptotic complexity. This optimization is the first solution of this particular performance problem.

4.2 Design

In this section, I discuss the design of orthogonal persistence for Sidney. Of the operations introduced in Chapter 1 only `alloc` has not been discussed. The use of orthogonal persistence means that determining whether an object is persistent requires a reachability analysis. To avoid the need for such analysis at allocation time, all allocations assume that the data are transient. This assumption means that the programming language's implementation of `alloc` is unchanged, an advantage since in SML/NJ allocation is very inexpensive. If an object is persistent, then after a `commit` its state should be saved on disk. Thus `commit` has primary responsibility for maintaining object persistence. The implementation of `commit` is the focus of the remainder of this section.

4.2.1 Basics

Figure 4.1 shows how a transitory heap is added to the design of the previous chapters. The persistent heap, shown in gray, represents the volatile and stable versions of from-space and to-space from the previous chapters. This figure emphasizes that the role of the transitory heap is to store objects that are not reachable from the persistent root. The transitory heap has its own root that is stored in volatile memory. All objects are initially allocated in the transitory heap. `Commit` uses the Transaction Manager to discover which transitory objects are persistent and should be moved into the persistent heap.

Figure 4.2 shows a detailed view of the key components of the system when it is in a committed state. In a committed state all objects reachable from the persistent root must be found in the persistent heap. Thus, in a committed state, no pointers may point from the persistent heap into the transitory heap. Pointers may point from the transitory heap into the volatile image, as is shown. Note that the volatile image is identical to the stable image.

Figure 4.3 shows the system when uncommitted data are present. `Writes` have created pointers from the volatile image into the transitory heap. `Commit` must guarantee that any object that is now reachable from the persistent root is in the volatile image, and that the stable image is atomically updated to reflect all changes to the volatile image.

All pointers from the volatile image into the transitory heap are assumed to refer to objects that are now reachable from the persistent root. This assumption is conservative: if objects that are not reachable from the persistent root are made persistent, the persistent collector will eventually discover this and make them transitory again or, if they are now unreachable, collect them. Since the persistent roots only come into play during GC, for simplicity I have omitted them from these figures.

During a `commit`, Sidney traverses the write log to identify pointers into the transitory heap and uses them as the roots of a copying garbage collection. I call this collection the *commit GC*. The commit GC moves all objects that are reachable from these roots into the volatile image. `Commit` then updates the stable image to include these newly persistent objects, as well as updating any existing objects that have been modified. Figure 4.4 shows the state of the system after the collection has been done and the stable image updated. The copy of the newly persistent datum is shaded; a forwarding pointer, shown as a dotted line, connects the original version in the transitory heap to the persistent version.

The cost of updating the persistent heap has three components. First, there is the cost of using the write log, both as roots and to log the entries to the low-level transaction manager; this cost is proportional to the number of `writes` in the log. Second, there is the cost of the commit GC that copies the newly persistent objects into the persistent heap; this cost is proportional to the amount of data that is copied. Finally, there is the cost of updating the stable image. This last cost has components proportional to the number of `writes` in the log and the amount of newly persistent data, but also has some fixed overhead that, since it includes a synchronous disk write, may well dominate the other terms, especially for small transactions.

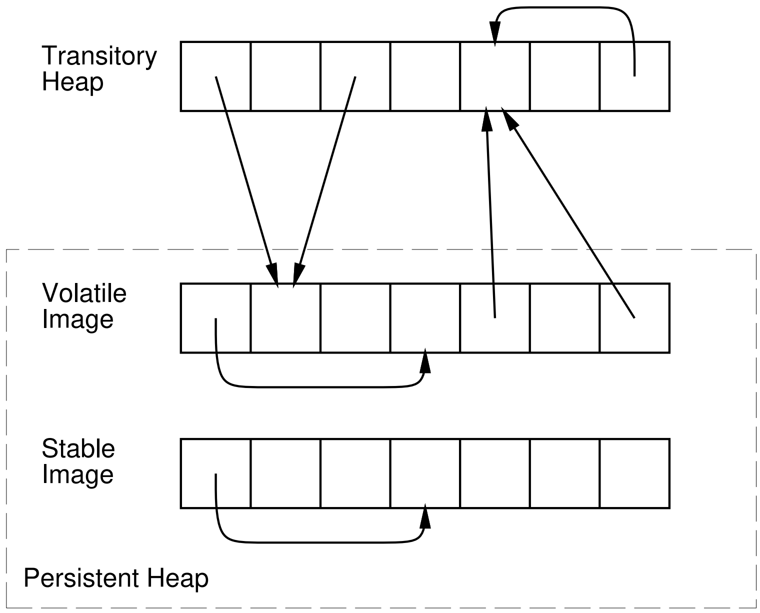


Figure 4.3: Before Commit

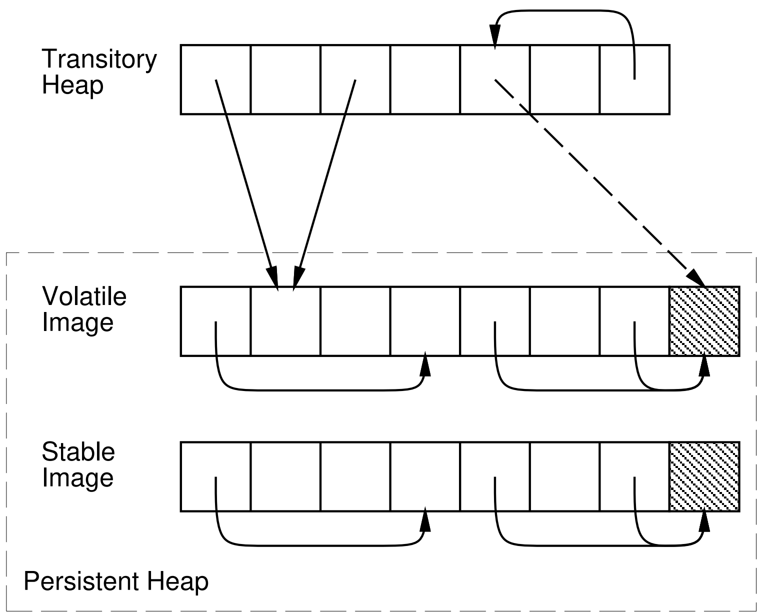


Figure 4.4: After Commit

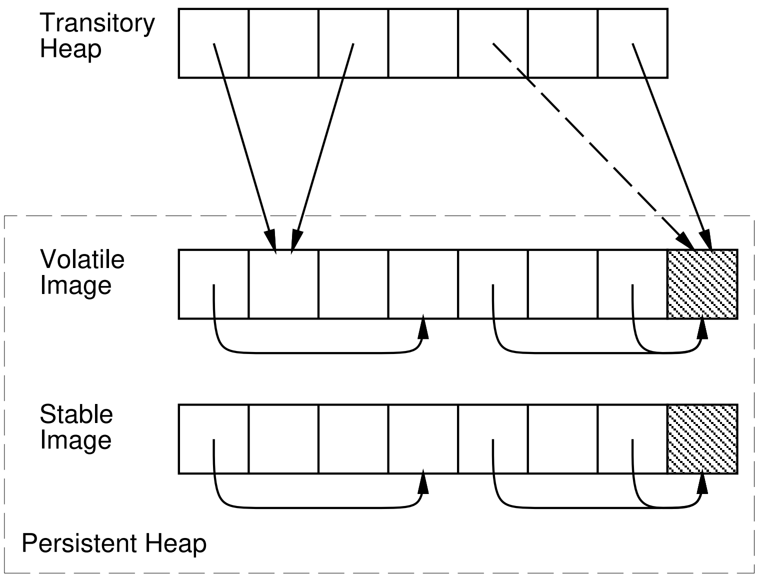


Figure 4.5: After Scanning

4.2.2 The Problem

The work of `commit` is not yet complete. Figure 4.4 also shows that pointers may remain in the transitory heap that refer to objects that have been moved into the volatile image. These are the pointers in the transitory heap that point to objects that have been moved and thus have forwarding pointers. These pointers must be updated to refer to the volatile image versions. A way to ensure that all such pointers are updated properly is to apply `scan` to the entire transitory heap, looking for such pointers and correcting them. The forwarding pointers left by the copying collection allow these pointers to be identified and updated. In this case, `scan` only serves to update pointers to objects that have been moved so that they point to the new versions, and does not force any objects to be `copied`. Figure 4.5 shows the result of such scanning. Another option is to garbage collect the transitory heap immediately; during the collection, the forwarding pointers will cause these references will be redirected to the copies in the volatile image.

Both of these methods add a cost to `commit` that is proportional to the size of the transitory heap. Unfortunately, it seems impossible to selectively track the pointers that will require updating on a `commit`. This problem exists because a `write` could make any transitory object persistent; thus tracking the pointers to potential persistent objects would require tracking all pointers to all objects. This is not practical. Even if it were possible to track the pointers, the number of them could be proportional to the size of the transitory heap. It seems inevitable that the cost of updating these pointers will depend on the size of the transitory heap. However, ideally, the latency of an individual `commit` should depend only on the number of `writes` performed and the amount of data that must be transferred to the volatile and stable images, and should be independent of the amount of transitory data.

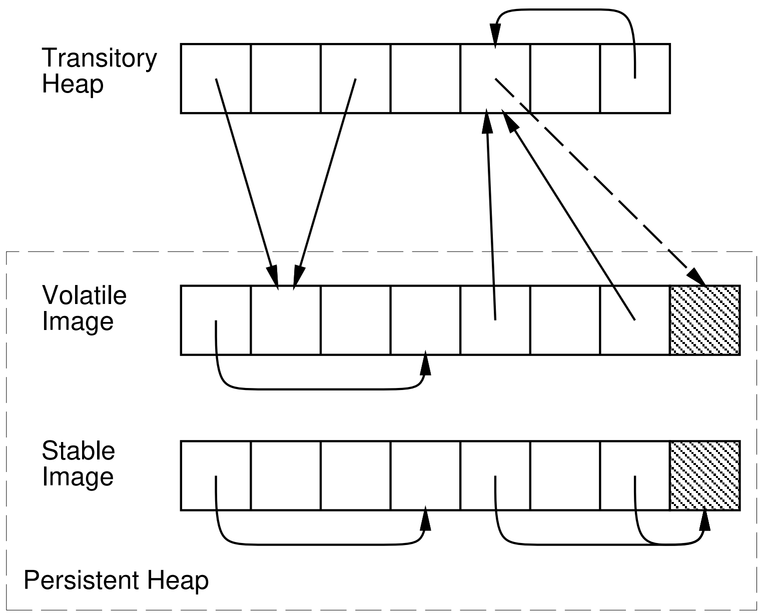


Figure 4.6: After Rollback

4.2.3 The Solution

If `commit` does move objects from the transitory heap into the volatile heap, then pointers will be left behind that, given the stated performance goals, are impossible to correct. Therefore, any solution to the problem must eventually leave these objects in the transitory heap. The key insight is that the semantic requirement of `commit` is that the stable image must contain the committed state. There is no fundamental requirement that the stable and volatile images be identical, nor that the transitory heap be updated. However, if updating the transitory heap pointers is delayed, then updating the pointers in the volatile image that caused these objects to be persistent must also be delayed. This observation suggests the following `commit` strategy:

1. Do the `commit` up to the point of correcting any invalid transitory heap pointers, but retain enough information to rollback the effects of this step.

2. Update the stable image using the volatile image. This leaves the stable image in the correct state.

3. Rollback the effects of the first step on the volatile image. This leaves the transitory heap in the correct state.

There are several problems with this strategy. One problem is that typically copying collectors destroy the original version of the object that they are copying by overwriting it with a forwarding pointer. However, it is not enough simply to repair this damage during rollback. The additional problem is that subsequent commits will need to modify the stable image in a manner consistent with the current placement of objects in the stable image. Thus, it is necessary to retain complete placement information in the volatile image.

Replicating collection can be used to solve these problems. Commit GC nondestructively creates replicas of the newly persistent objects in the volatile image. `Commit` also updates the volatile image to refer to the newly persistent objects as needed. When the stable image is updated, these replicas are considered to be the "real" versions. Rollback then changes the relevant pointers in the volatile image to refer back to the original objects in the transitory heap, leaving the new objects to act as "to-space replicas." This approach both avoids destroying the original objects and leaves the replicas in the volatile image, thus preserving the placement information.

Figure 4.6 shows the state of the system after the rollback step. Note that the stable image is in the same state as it is in Figure 4.4 after a `commit`. However, the two pointers that cross between the volatile image and transitory heap have been restored, leaving the transitory heap version of the object as the primary copy. The secondary copy has been left in the volatile heap and the forwarding pointer has also been retained.

After `commit` has been completed, the client continues to use the original objects in the transitory heap. Although the replicating collection is non-destructive, it does leave forwarding pointers to the objects it copied; it simply does not overwrite the original object with them. Subsequent `commits` will not recopy these objects because they are already marked with forwarding pointers. The write forwarding used by replicating collection will ensure that the replicas are kept up to date.

Later, when the system eventually garbage collects the transitory heap, all of the pointers in the transitory heap will be updated just as in Figure 4.5. At that time, the pointers in the volatile image that were reset to their original values will also be updated to point to the persistent version of the object. This update does not require a change to the stable image, because in the stable image these pointers already refer to the correct values.

4.2.4 Performance Ramifications

As already discussed, the cost of `commit` up to the point that the stable image has been updated has a cost that includes a fixed term, a term that is proportional to the number of `writes` in the log, and a term that is proportional to the amount of newly persistent data. Adding replication-based rollback requires that we keep track of enough information to do the rollback. All this requires is remembering the original values of the pointers that go from the volatile image to the transitory heap, and is thus at worst proportional to the number of `writes`. Doing the rollback itself just requires restoring these values, and so also is proportional to the number of `writes`. This fact means that adding rollback, which avoids the cost of scanning the transitory heap, does not change the asymptotic complexity of `commit`. Adding replication-based rollback has thus allowed us to reach our asymptotic performance goals.

4.3 Design Alternatives

In this section, I discuss alternatives to the use of orthogonal persistence. One obvious alternative is simply to make all data persistent. I have already argued that this is not practical. The principal alternative is to make persistence a property the programmer must deal with explicitly.

In retrospect, I believe that the use of orthogonal persistence is probably the least well justified aspect of Sidney's design, at least from the perspective offer by this dissertation. Originally, I chose to use orthogonal persistence because it fit well with the semantics and "feel" of SML. Since my original goal in designing Sidney was to support transactions and persistence in SML, I believe this was a correct decision. In the current context, I believe that much of the value of adding orthogonal persistence lies in exploring whether it is possible to implement it efficiently. If it is, then designers are free to use it without penalty if they believe it is useful.

4.3.1 Explicit Persistence

The most obvious way to specify explicitly which objects are persistent is to provide a special allocation routine for persistent data. Some of the disadvantages of this approach are related to programmer convenience, but some relate to safety. If the persistent heap contains committed references to volatile objects, then upon recovery these pointers can not refer to valid data, since the transitory heap they point into no longer exists. The presence of pointers to invalid data compromises data integrity.

An important advantage is that using a special allocator is very simple for the implementor. It is less clear that performance is an advantage. Asymptotically, an explicit system will still have overhead associated with each change to the persistent heap, with the newly allocated data, and with updating the stable image. The performance evaluation in Part II, will shed some light on the constant factors involved.

Another approach is to tie persistence and data type or class together [47]. The term "orthogonal persistence" comes from the fact that in such a system persistence is orthogonal to other properties of the object, and, in particular, to its type. In a type-based persistence system, the problem of dangling pointers can be dealt with by not allowing persistent data types to refer to transitory data types. However, this approach has other problems. The biggest problem is that it requires that the programmer code different implementations for persistent and non-persistent data, even if the representations are the same.

Chapter 5

Implementation

The design presented in the previous chapters is relatively high-level. In particular it is independent of the particular programming language and language implementation to be used, as well as the low-level transaction manager. It does depend on the language implementation being compatible with copying collection. In this chapter, I discuss the details of Sidney's implementation. Understanding these details is not essential to understand the basic design, but will help when reading the performance evaluation.

The implementation presented here is the first of its kind. One previous implementation [1] was done of a concurrent mark-and-sweep collector for a persistent heap, but that implementation did not support transactions. Two other designs for concurrent copying collectors of transactional persistent heaps have been proposed, but only resulted in partial implementations. All of these are discussed in the related works, Chapter 11.

The chapter begins by discussing some other systems on which the implementation depends. An overview of the implementation follows. Next, there is a discussion of the implementation of the parts of Sidney directly executed on behalf of the user code, including `commit`. The description of the collector begins first with a description of the implementation of concurrent replicating collection for transitory heaps and second, with a discussion of stop-and-copy collection of the persistent heap. Finally, the chapter ends by combining these two designs and showing the complete implementation.

5.1 System Preliminaries

The implementation relies on two pre-existing components, Standard ML of New Jersey (SML/NJ) and Recoverable Virtual Memory (RVM). SML/NJ is the programming language implementation on which Sidney is based. RVM provides the ability to update the disk efficiently and atomically. The next two subsections discuss the relevant aspects of SML/NJ and RVM.

5.1.1 SML/NJ

Standard ML (SML) is a strongly, statically typed programming language that supports first-class functions and a sophisticated module system. Programmers typically use SML in a "mostly functional" style that discourages the use of assignments. Strong typing combined with garbage collection and array bounds checking provides the programmer freedom from the kind of pointer-related errors that lead to using invalid or illegal addresses. This means that SML never "dumps core" making it safer and easier to program in than most conventional languages.

This dissertation grew out of an effort to add transactions to SML [23]. Much of the original motivation for the use of implicit persistence techniques was to create a persistence system that fits well with SML's language model. In particular the emphasis on safety was directly inspired by SML.

Sidney's implementation is based on Standard ML of New Jersey (SML/NJ) [2] (version 0.75). SML/NJ is an optimizing compiler for SML. The SML/NJ source code is freely available and is easy to modify for experimental purposes. The runtime system of SML/NJ is typical of runtimes that support copying garbage collection, and the results presented here should transfer easily to language implementations for variants of Lisp [46, 54], Haskell [27], Modula-3 [40], etc. The version of SML/NJ being used has been extended to support client threads and multiprocessors [37].

For this dissertation, an important aspect of SML/NJ is its support for generational collection. The SML/NJ heap is composed of two spaces, *new* and *old*, as shown in Figure 5.1. All allocation occurs in the new space. During allocation a limit check tests to see if there is space left in the new space. If there is not enough space, an arithmetic trap causes a transfer into the garbage collector. The collector then copies all of the live data from the new space to the old space; this collection is called a *minor* collection. If a minor collection causes the old space to become full, then a standard two-space copying collection is done; this collection is called a *major* collection. Minor collections usually copy less data than a major collections and thus have lower individual cost. However, they are much more frequent and so tend to be the dominate GC cost, despite the name "minor".

SML/NJ must track pointers from the old space into the new space so that they may be used as roots during the minor collections. Such pointers are only created by `writes`. SML/NJ tracks these assignments by using in-lined code to add the location of each `write` to a list, called the *storelist*. The original version of SML/NJ only adds writes to pointer-containing locations to the storelist, since only these `writes` are needed for generational collection. The storelist forms the write log discussed in the design chapters. I will continue to use the term "write log" here rather than "storelist".

SML/NJ is unusual in that it does not use a stack. Instead, all the procedure call frames are allocated on the heap. This causes the allocation rate to be very high, about one byte every five instructions. Allocation is highly optimized, taking only a little more than one instruction on average, and adding any overhead to allocation would have unacceptable performance consequences.

Another unusual aspect of SML/NJ is that most of the code it executes is stored in the heap and not in the text segment. This is atypical of most similar language implementations,

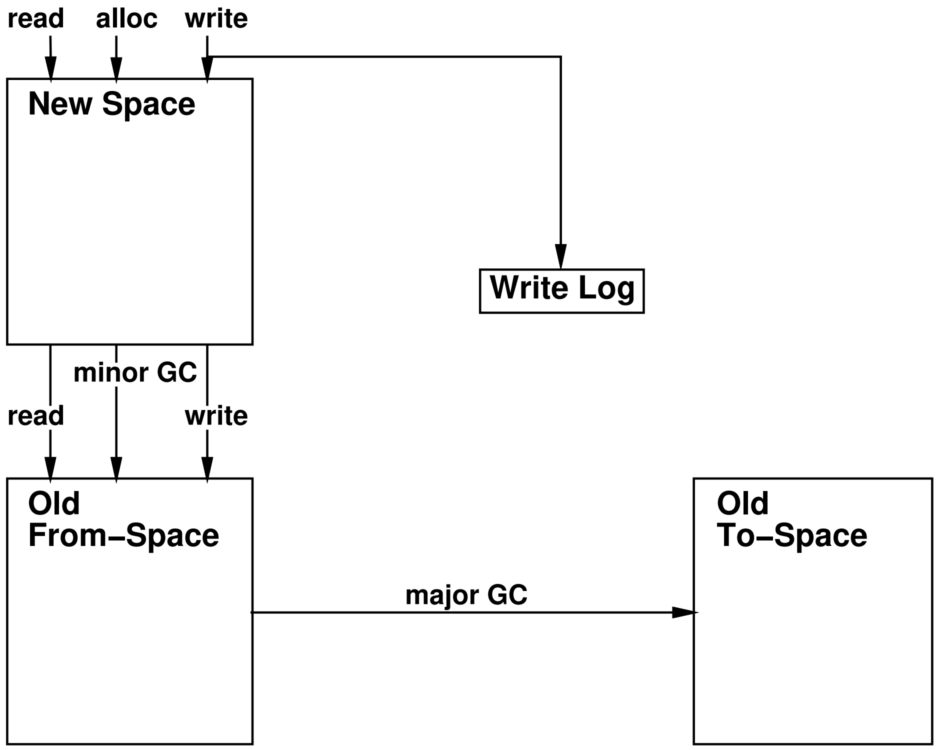


Figure 5.1: SML/NJ Heap Organization

and the reasons for doing so are irrelevant. An important consequence is that the garbage collector may move the code, which requires that the instruction cache be flushed after each collection; the flush occurs even if no code is actually moved. A better solution would be to flush the cache only when code is moved, but unfortunately, SML/NJ has no way of reliably detecting when this happens. This issue is of special importance to Sidney because, in general, the commit GC requires a flush.

5.1.2 Recoverable Virtual Memory

Sidney's design assumes the ability to make multiple updates to the stable image both atomically and efficiently. For this purpose, Sidney's implementation uses the Recoverable Virtual Memory (RVM) system described by Satyanarayanan et al. [51]. RVM provides simple non-nested transactions on ranges of bytes in virtual memory. Sidney's implementation does not exploit the rollback features of RVM at all; RVM is only used for disk logging and recovery.

The key abstraction provided by RVM is the *segment*. Segments are portions of files (or raw disk partitions) that can be mapped one-to-one with ranges of virtual memory. RVM defines a simple interface that allows changes to segments mapped into the virtual memory image to be transferred to disk atomically. The interface provides the ability to begin RVM transactions, to commit RVM transactions, to abort RVM transactions (not used by Sidney), and to record (or log) that a range of bytes in a segment has been modified. RVM guarantees that the modifications logged between a begin transaction and an RVM commit are transferred to disk atomically.

RVM achieves both efficiency and atomicity of disk writes by using a disk-based log. Rather than writing updates directly to their location in the disk file containing the modified segment, updates are appended to the log. Atomicity is achieved by using special log records to indicate the beginning and end of atomic sequences. Efficiency is enhanced because all changes are recorded in a single write and because log writes are sequential. If the disk arm has not been moved by another process, sequential writes will generally not require a seek although they may incur rotational delays. When the RVM log becomes full, it must be *truncated*. Truncation forces all of the changes in the log to be applied to the actual data on disk.

5.2 Overview of Implementation

Figure 5.2 gives an overview of the implementation. The basic operations, `read`, `write`, `commit`, and `alloc` are shown on the top of the diagram. The transitory heap is just SML/NJ's standard heaps; the persistent heap is basically the same as in the design, except for some additional details that will be discussed shortly. Boxes represent the basic data structures of Sidney, while circles represent code that executes on behalf of the various threads in the system. Boxes shown in gray are located on disk, while all others are in volatile memory. The implementation uses three threads: the I/O thread that is responsible for transferring the volatile to-space to disk, the GC thread that replicates reachable objects in to-space, and the client thread that executes user code, performs `commit`, and initiates and terminates garbage collections.

`Read` and `write` use the from-space version of objects. `Writes` are logged, including enough information to allow the `writes` to be undone. All allocation occurs in the new space and `alloc` is unchanged from the original SML/NJ implementation. For minor collections, old from-space acts as new to-space. Although not shown explicitly, replicating collection may (optionally) be used for minor and major collections. Doing so requires that `writes` to these heaps also be logged.

The transaction manager executes as part of the client thread during `commit`. The transaction manager reads the write log to determine the roots for the commit GC, as well as to discover what locations must be updated in the stable from-space image. The commit GC transfers data from the transitory heap into the volatile from-space image. The newly persistent data as well as the new values of all the modified locations in the volatile from-space image are transferred to stable storage using RVM. After the stable from-space image is updated, to avoid having invalid pointers in the transitory heap it is possible to use one of three strategies: replication-based rollback, garbage collecting the transitory heap, or `scanning` the transitory heap to fix invalid pointers.

Garbage collections are triggered when the amount of data allocated in a from-space exceeds a parameterizable limit. Minor collections are triggered by `alloc`, major collections during minor collections, and persistent collections during `commit`. If replicating collection is enabled then the collector records the current root set and allows the client thread to re-

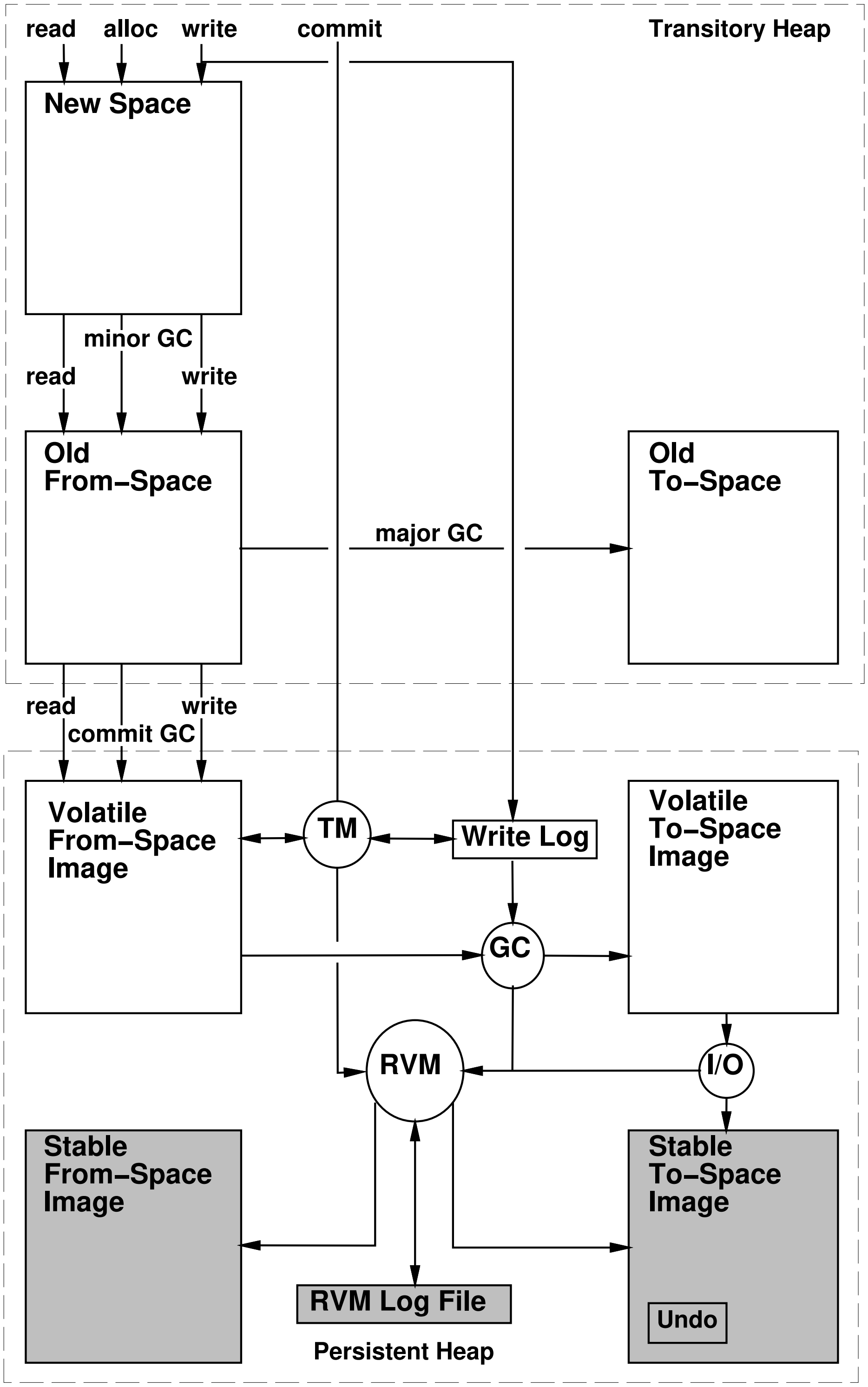


Figure 5.2: Implementation Overview

sume. The GC thread then replicates the objects reachable from the roots concurrently with the client thread. When the GC thread has replicated all the live objects, it interrupts the client thread to obtain the current root set as well as the write log. During this interruption, the client thread performs a bounded amount of garbage collection work in an effort to achieve the flip condition. If the collector is able to flip, then the collection terminates. If the collector does not achieve the flip condition then the GC thread is resumed, while the client thread returns to executing user code. The collector shares the write log with the transaction manager.

Since persistent GC must guarantee that the persistent heap is recoverable, the conditions it must satisfy to flip are more extensive than for transitory heap collection. The most obvious additional requirement is that the volatile to-space image must be written to stable storage. The I/O thread writes the volatile to-space to disk, either directly or using RVM. The collector must also write the flip record indicating which disk file represents the current stable from-space. The flip record is written using RVM, but without forcing the record synchronously to disk. If a crash occurs before the flip record actually reaches disk, recovery will use the old from-space, which is perfectly acceptable. Thus, a feature of this design and implementation is that persistent GC requires no synchronous writes to disk. The final requirement to achieve the flip condition is that the undo log must also be included in the stable to-space.

The rest of the chapter elaborates the basic points above in detail. The goal in describing them is both to allow others to replicate the implementation and to provide a basis for understanding the detailed performance evaluation in Part II.

5.3 User Code

This section describes those portions of the implementation that execute directly on behalf of the code written by the user. These include several modifications to the compiler as well as the implementation of `commit`. `Read` and `alloc` are unchanged by Sidney's implementation and thus suffer no performance penalty from Sidney.

5.3.1 Compiler Modifications

I modified the SML/NJ compiler to include the additional write logging needed by the design and to allow a space-related optimization of replicating collection.

Write Logging

The write log supports generational collection, commit GC, replicating collection, updating the stable from-space on `commit`, persistent collection, and `abort`. Generational collection requires the logging of all `writes` that create pointers from old-space into new-space. Commit GC requires logging all `writes` that create pointers from the persistent heap into the transitory heap. Replicating collection requires logging all `writes` to any heap collected concurrently. Updating the stable to-space requires logging all `writes` to the persistent heap. Persistent collection requires logging undo information for all `writes` to the persistent heap. Finally, although it is not carefully considered here, `abort` requires logging undo information for any `write` that must be rolled-back during abort.

SML/NJ already logs all pointer `writes` using the storelist to support generational collection. I extended this support to include every `write` and to save the values being overwritten for rollback. The details of these modifications are not important, although the following points are significant. The cost of logging `writes` depends on the kinds of data being written; the possible kinds are pointers, integers, arrays of bytes, and floating point numbers. Logging undo information incurs an additional cost. The actual logging is done by in-line code generated by the compiler. Typical sequences are ten instructions long and include five writes. This implementation is not optimal; using a store buffer instead of a list makes sequences of four instructions with two writes feasible. Conceptually this is a simple change, but it is likely to be hard in practice, since the current list representation is deeply embedded in SML/NJ's implementation.

Header Merging

Any copying garbage collector must maintain a relocation map that translates the location of a from-space object to that of its to-space version. It is important that this mapping be both compact and inexpensive to access. Typically this is achieved by using forwarding pointers that are stored with the from-space object and that point to the to-space version. These forwarding pointers are created at the time the object is first copied into to-space, and typically they overwrite the from-space version, destroying it. Replicating collection requires that the from-space version be left intact.

An easy way to avoid overwriting the object would be to reserve an extra word at the beginning of an object to be used by the collector. However, most objects in the SML/NJ heap are only three words long and already include one word of header information. Thus, adding an additional word for a forwarding pointer would be too costly in space. Instead, after copying the header word to to-space, Sidney overwrites the header word found at the beginning of all objects with the forwarding pointer.

The header word consists of 4 bits of tag information and 28 bits representing the length of the object. Since it must be possible to distinguish forwarding pointers from normal headers, the forwarding pointer must also be tagged. Objects are located on word boundaries so only 30 bits of the forwarding pointer are useful. By sacrificing an additional 2 bits, forwarding pointers can be stored in the upper 28 bits of the header and tagged in exactly the same way as any other header. This restricts the implementation to heaps containing less than 2^{30} bytes, which has proven acceptable in the current implementation.

Overwriting the header word implies that any operation that accesses the header word must be able to detect and follow the forwarding pointers. The user accesses the header word only when executing the polymorphic equality operator and array and string length operations, all of which occur infrequently. As shown in Figure 5.3, the client operations that read the header word follow the forwarding word, if it is present. However, in the presence of concurrency, this change creates a potential read-write conflict between the collector and the client. If the client is reading the header word at the same time the

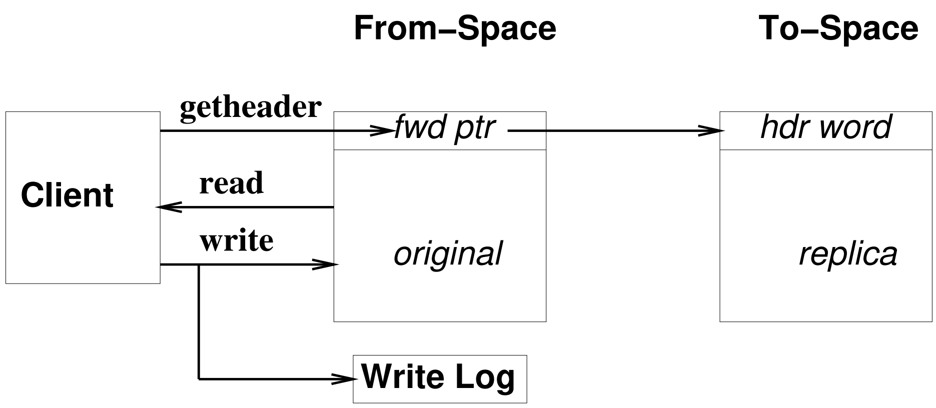


Figure 5.3: Getheader Operations Follow the Forwarding Word

collector is installing a forwarding pointer, the implementation must make sure that the client finds the correct header word.

The code sequences used by the client and the collector for these operations were designed to avoid any possible race condition. The client reads the from-space header word only once and then dereferences the value obtained if it is a forwarding pointer. The collector replaces the from-space header word with the forwarding pointer only after storing the correct header word in the to-space replica. This method works properly provided that the memory system performs single-word write operations atomically and that several write operations issued from a single processor are performed in the order issued.

Replication-based rollback creates the possibility that chains of forwarding pointers of up to length two may occur. This case arises when a transitory heap object has been copied into the volatile from-space during a `commit` and then the volatile from-space version is replicated into volatile to-space. I further modified both the compiler and the collector to allow for such chains.

Merging the forwarding and header words is not fundamental to the technique of replicating collection. In systems that have larger average object sizes, it would be perfectly acceptable to add an extra forwarding word. Doing so would both eliminate much of the cost of following forwarding words, and more importantly would reduce the complexity of the implementation.

5.3.2 `Commit`

`Commit` is the most complex of the client operations. `Commit` is responsible both for maintaining the correct state of persistent objects on disk and implementing orthogonal persistence. Conceptually, this is just a matter of performing the following operations that have been discussed previously:

- Begin an RVM transaction.
- Perform a commit GC.

- Record `writes` to the persistent heap with RVM.
- Record newly persistent data with RVM.
- End the RVM transaction.
- Fix invalid transitory heap pointers.

Implementation Details

In practice, the implementation is more complicated. The following is an enumeration of all the important steps (omitting, for example, function call boundaries) in the order in which they occur, along with an explanation of what purpose they serve. The starred steps are the ones from the list above. The items marked "required" are required by the design or aspects of the implementation not under my control. The items marked "implementation convenience" could be eliminated, but at the cost of complicating the implementation.

1. Pause the GC thread if active. Any concurrent copying of data would complicate commit GC. (Implementation convenience.)
2. Do a minor collection. The presence of two transitory spaces would complicate commit GC. Performing a minor GC, or completing one in progress, eliminates one of the heaps. (Implementation convenience.)
3. Finish any active major collection. The presence of transitory replicas would complicate the commit GC. Completing any pending major collection eliminates any such replicas. (Implementation convenience.)
4. (*) Begin an RVM transaction. RVM requires that a transaction be open to allow modifications to be recorded. (Required.)
5. (*) Perform a commit GC. The commit GC moves all newly persistent data from the old space to the volatile from-space. (Required.)
6. Forward `writes` to the persistent heap to replicated objects. Forwarding `writes` at this point is inexpensive and allows the write log to be discarded after `commit`. This simplifies the implementation. (Implementation convenience.)
7. (*) Log `writes` to the persistent heap to RVM. Any location in the persistent heap that has been modified must have its new value recorded on disk. (Required.)
8. Discard write log and record discard with RVM. The write log can be discarded, since it is no longer needed for `commit` and write forwarding. Logging the discard has the effect of canceling the undo log in stable storage. (Required.)
9. (*) Record newly persistent data with RVM. The data that was added to the volatile from-space image by the commit GC must be recorded in the stable from-space image. (Required.)

10. Record the location of the new end-of-heap with RVM. (Required.)
11. (*) End the RVM transaction. Committing the RVM transaction forces all updates synchronously to disk. (Required.)
12. (*) Fix invalid transitory heap pointers. Invalid transitory heap pointers must be corrected using one of the three techniques discussed in detail below. (Required.)
13. Trigger persistent heap collection if needed. `Commit` is the only way that new data are added to the persistent heap. Thus, persistent heap overflow is detected at this point. (Required.)
14. Flush instruction cache. SML/NJ stores code as well as data in the heap. Consequently any operation that copies data must flush the instruction cache. (Required.)

Transitory Heap Pointer Correction

An important issue is how to correct the invalid transitory heap pointers created by the commit GC. The implementation supports three different strategies: garbage collecting the transitory heap, `scanning` the transitory heap, and replication-based rollback. The first two of these are very simple to implement but have poor asymptotic complexity, while replication-based rollback offers good asymptotic complexity but complicates the implementation.

The simplest way to correct invalid pointers is to garbage collect the transitory heap. Since the objects referred to by the invalid pointers have been copied and thus have forwarding pointers, any references to them will be updated appropriately during collection. Implementing this option is trivial since it must already be possible to garbage collect the heap. The disadvantage is that collecting the transitory heap is potentially a very expensive operation.

The second option is scanning the transitory heap. `Scan` is usually applied to objects in to-space to update their from-space references to the to-space version (of course, it also serves to force any uncopied objects in from-space to be copied). When applied to the objects in the transitory heap after a commit GC, `scan` will update any references to objects that are now located in the persistent heap. Almost all of the code needed to perform this `scan` is already present, making the implementation straightforward. The disadvantage of this approach is that it requires examining every object in the transitory heap and applying `scan` to every location that might contain a pointer.

Support for replicating collection, notably non-destructive copying and write-logging and forwarding, also provides the basic mechanisms needed to support replication-based rollback. However, this feature still requires modifying a number of parts of the system, including `commit`, write forwarding, and how the transitory and persistent heaps are flipped.

To support rollback, we need to make the following changes to `commit`.

- When `commit` updates a root pointer, the pointer's location and old value are saved in a log. This *commit log* has the same format as the write log already in use by the system.

- When `commit` has updated the stable image, the commit log is used to restore the pointers into the transitory heap. The location of the pointers that have been rolled-back are saved in another log, the *rollback log*. When a major collection occurs, the rollback log is used to find the locations of persistent heap pointers that must be redirected to their persistent heap versions.

The additions needed to other parts of the implementation to accommodate replication-based rollback are covered in the sections concerning garbage collection. Sidney supports all three techniques outlined above.

5.4 Garbage Collection

In this section, I describe, in three steps, the implementation of the garbage collector. First, I discuss the implementation of concurrent replicating collection for transitory heaps. Second, I discuss the implementation of stop-and-copy collection for persistent heaps. Finally, I show how to combine these to yield a concurrent collector for persistent heaps.

5.4.1 Concurrent Replicating Garbage Collection

Even without application to persistent heaps, replicating collection is a useful, new approach to concurrent copying garbage collection. In this subsection, I discuss how replicating collection is implemented for transitory heaps, ignoring all complications that arise when the persistent heap is added. For more details about replicating collection for transitory heaps, including performance results, see [41, 43].

Two classes of issues arise in adding replicating collection to SML/NJ. The first class involves support for the basic mechanisms used by replicating collection. The second class concerns the policies needed to coordinate the collector and the client. These latter issues are not unique to replicating collection and must be dealt with in some form by any concurrent collector.

Mechanisms for Replicating Collection

The basic mechanisms used to support replicating collection are non-destructive copying, logging of `writes` to from-space objects, and forwarding `writes` to to-space objects. The details of non-destructive copying have already been covered because of their impact on the `getheader` operation.

Most of the details of adding `writes` to the log have been covered. In addition to the logging done by the compiler, any `writes` done by the runtime to SML/NJ objects must also be logged. These modifications occurred almost exclusively in the runtime code that supports I/O.

`Writes` that have been logged must be forwarded to their to-space replicas. Any object that has a to-space replica will have a forwarding pointer in its header word. When the write log is processed, the objects that have been written are tested for the presence of a forwarding pointer. If a forwarding pointer is present then the current value is read from the

Parameter	Function
N	Bytes of new space allocation before minor collection.
O	Bytes added to old space before major collection.
P	Bytes added to persistent heap before persistent collection.
L	Maximum bytes copied by client thread before returning to user.
U	Maximum bytes copied by GC thread before polling.

Table 5.4: GC Parameters

from-space version and written to the to-space replica. Even if the client is writing to the from-space version concurrently there is no conflict, since the new `write` will be logged in the usual way and the correct value forwarded when the new log entry is processed. One subtle point is that if the to-space replica has been scanned, the new value must also be scanned. This maintains the invariant that any object below the scan pointer contains only to-space pointers.

One other mechanism is needed for replicating collection. When a root is scanned in a stop-and-copy collector, not only is the object it points to copied, but the root itself is updated to point to the new version. In replicating collection, the roots must be used to find reachable objects before the flip, but only at the flip should they be updated. I added new versions of the relevant routines that do not update the root.

Policies for Concurrent Collection

In a system based on stop-and-copy collection, when a garbage collection is triggered, the collector runs while the client is stopped, and when the collector finishes the client resumes. The asynchronous nature of concurrent collection requires more coordination between the collector and the client. One important advantage of replicating collection is that no low-level synchronization is needed between the collector and the client, but coordination is still needed to initiate collections, to get the current roots, and to flip the heap.

To control when garbage collections occur, Sidney uses three parameters (Table 5.4), N, O, P, each of which specify the number of bytes that may be added to a particular space before a garbage collection is triggered. When N bytes have been allocated in new space, a GC trap occurs and a minor collection is initiated. When O bytes have been added to old space since the last major collection, a major collection is initiated. When P bytes have been added to the persistent heap by `commit`, a persistent collection is initiated. There are also parameters that control whether stop-and-copy or concurrent collection is used for each space. If stop-and-copy collection is enabled on a space, then the collection is done completely when it is initiated.

Eventually, an ongoing concurrent collection will have copied all the data accessible from the roots the collection started with. Meanwhile, the client will have been creating new log entries, changing the roots, allocating new data, etc. The collector needs the write log and current roots to continue. It obtains them by interrupting the client thread. During this interruption, the client thread is allowed to do a bounded amount of work on behalf of the collector. If, during this period, the flip condition is reached, then the flip is done and the collection terminates. If the flip condition cannot be met then the client thread restarts the GC thread and resumes work on client code. The amount of work that is allowed to be done during the pause is controlled by a parameter (Table 5.4), L, which specifies the number of bytes that can be copied before giving up and resuming the client.

Another synchronization issue is when the client thread must interrupt the GC thread. Instead of using a signaling mechanism, the GC thread simply polls an interrupt flag periodically. A parameter (Table 5.4), U, controls the number of bytes that the thread copies before polling.

The final issue is when the write log is processed. In theory, much of the work associated with the write log could be done concurrently. However, the log is stored in new space, and since new space is reused after each minor collection, Sidney processes the log while the client thread is paused for minor collections.

5.4.2 Stop-and-Copy Persistent Garbage Collection

This section presents the additions to a conventional stop-and-copy collector needed to collect the persistent heap. Two classes of issues arise. The first class concerns making the heap recoverable, for example writing the volatile to-space image to stable storage. The second class arises because there are now two heaps, and the pointers between the two heaps must be maintained correctly when flipping. The latter becomes a particularly important issue with the addition of replication-based rollback.

Making the Heap Recoverable

The actual process of copying volatile from-space to volatile to-space is unchanged due to the addition of persistence, but there are two additions needed to make the heap recoverable. First and most obvious is that the volatile to-space must be transferred to stable storage. Second, the undo log must be written to stable storage. At this point, it is also important to understand how the heap is represented on disk.

Sidney stores the stable heap images in two files, one for stable to-space and one for stable from-space. Each of these files contain two segments, one of which contains the location of the beginning and end of the heap as well as other heap meta-data, and one of which contains the actual heap. The segment containing the heap bounds may be mapped anywhere in memory, but because it contains pointers, the segment containing the heap must be mapped into the same locations as it was on previous uses of the heap. If this is not possible, Sidney can examine the heap data and correct the pointers so that they reflect the new location of the heap. This is feasible because all pointers can be identified and adjusted by the garbage collector. The persistent root is the first location in the heap. Usually this location then points to a symbol table, so that the user may add persistent data to the heap in a way that lets it be looked up by name; the symbol table and lookup mechanisms are part of the higher-level transaction system built on top of Sidney [23].

Writing the volatile to-space to stable storage is the most important addition needed to support recoverability. The obvious approach is to just write the heap directly to a file. This approach introduces a problem with RVM. RVM uses its disk log to hold updates that are eventually applied to the files that contain the heaps. The RVM log may contain writes that are destined for the same file as is being written directly. The writes in the RVM log are "stale," and if they are applied to the file they will corrupt the heap. Thus, after writing the heap it is necessary to ensure that this problem cannot happen. The solution is to cause RVM to truncate its log as part of the flip. Truncation forces all updates to the files containing the segments to which they apply. As long as the RVM log is truncated after the last user updates to a file and before the file is reused, the RVM log will never contain invalid data.

As an aside, a useful addition to RVM would be enabling selective deletion of obsolete records from the log. This feature would avoid the need for truncation. In general, the ability to inform other subsystems that large regions of memory are no longer needed or valid is a useful feature for copying garbage collectors. For example, significant speedups can be achieved in the presence of paging if the collector can inform the virtual memory system that certain pages can be dropped instead of being paged to and from the disk [18].

Since garbage collection is done on the volatile from-space image, which contains uncommitted data, the undo log must also be made recoverable before the persistent heap may be flipped. Since the log is an SML data structure itself, this is trivially accomplished by adding the root of the log to the collection roots.

One other issue relating to the undo log is when and if it is used. The undo log must be in stable storage as long as there are uncommitted data in the stable image. In the current system this is between the time of the flip and the next `commit`. The `commit` makes the log obsolete, so at that time the stable pointer to the log is overwritten with a nil value. Any garbage that is created in the persistent heap because of the need to stabilize the undo log will be collected during the next persistent heap collection.

For recovery, Sidney must be able to tell which file is the current from-space, so each file contains a sequence number. The highest sequence number identifies the current from-space. At the time of the flip this sequence number is incremented atomically by writing its new value using RVM. When this has been done, the roots are updated and the flip is complete both in memory and on disk.

The recovery process is very simple. First, Sidney opens the two disk representations of the heaps, which forces RVM to go through its recovery process, making the disk representation of the heaps up-to-date and emptying the RVM log. Next the sequence number is examined to find the current from-space. Finally, if the undo log is not empty, it is used to rollback any uncommitted data in the heap. The dissertation does not address the issue of the speed of recovery at all. The speed of recovery seems to be adequate in practice.

Cross Heap Pointers

Whenever a garbage-collected system has multiple heaps or spaces, pointers may cross between them. This issue is the reason generational collectors must log writes: the only cross-heap pointers that matter to them are those created by writing pointers to an older generation that point into a younger ones. The addition of the persistent heap creates the possibility of pointers between it and the transitory heap, and such pointers must be handled correctly, especially during flips.

Cross heap pointers must be tracked because they must serve as roots during garbage collections and must be updated during flips. We now consider in turn what kinds of cross heap pointers are possible. Pointers from the old space into the new space are tracked in the write log, and are used as roots during a minor collection and updated during a minor flip. Pointers from new space to old space are not tracked, because major flips always occur when the new space is empty. Pointers from the persistent heap into the transitory heap are also tracked in the write log. Persistent to transitory pointers are used as roots during minor, major, and commit GCs, and are updated when these collections flip. It would be feasible to track pointers from the transitory heap into the persistent heap, but the implementation does not do so. Such pointers must be used as roots during a persistent heap collection and updated at the time of a persistent heap flip. The implementation finds such pointers by examining all locations in the old heap.

Replication-based rollback adds considerable complications to the issue of cross-heap pointers. First, the pointers in the persistent heap that have been rolled back must be used as roots during transitory heap collections. These same locations must be redirected to point to their persistent replicas during a major flip. In addition, the locations pointed to by the rolled-back locations must be used as roots in persistent collections, but not updated during persistent flips. This requirement is because these locations are actually persistent objects that have only temporarily been left in the transitory heap. To complicate things more, rollback leaves forwarding pointers in the transitory heap that point into the persistent heap. These pointers must also be updated on a flip of the persistent heap. Currently this is done by `scanning` the transitory heap during a persistent heap flip.

Finally, what happens to objects that cease to be persistent? Since persistence is based on reachability, it is possible for objects to become transitory by virtue of becoming unreachable. If these objects are reachable from the transitory root, they are not garbage and must be retained. In principle, such objects could be moved back into the transitory heap. However, the current implementation does not do this. Instead, it simply leaves such objects in the persistent heap. If a crash occurs they will become completely unreachable and the next persistent GC will collect them.

5.4.3 Concurrent Replicating Persistent Garbage Collection

This section considers the additions needed to combine concurrent replicating collection with persistent collection to yield a concurrent replicating persistent collector. The implementation of concurrent replicating collection described above is sufficient to allow volatile from-space to be copied concurrently. The major modification needed to make a high performance collector is to make the process of copying the heap to disk concurrent.

Concurrent Heap Stabilization

Since the collector is concurrent and does not require that the client wait until all collection activity is finished, the process of copying to-space to disk can be made concurrent as well. One possibility would be to let the GC thread do all of the I/O in parallel with the client. This approach will sacrifice some concurrency since, during I/O, the GC thread will block. Instead the implementation adds a new thread, the I/O thread, to handle all blocking I/O. In addition to writing to-space, the I/O thread also performs the RVM log truncation. The truncate must complete before another flip can be initiated, but otherwise can be done completely concurrently with the other threads.

Getting good performance for writing to-space is more than just a matter of adding a new thread. When the I/O process is concurrent, it does not suffice to make getting the data to disk as fast as possible; what is important is minimizing the time the client is paused during the flip. This leads to a reevaluation of the techniques for writing data to disk. Two possibilities present themselves: we can write volatile to-space directly to the file containing stable to-space, or indirectly via RVM.

During the garbage collection, large contiguous regions of new data are created in volatile to-space. These regions can be written directly to disk efficiently. Logging them through RVM is less efficient because RVM first writes them into its log and then later rewrites them to the data file.

However, the garbage collector also performs small random updates on volatile to-space when it uses the write log to update inconsistent to-space replicas. RVM is ideal for applying these changes to stable to-space, because its use of logging converts the small random writes into a single efficient RVM log write. Writing these small random changes to disk directly is quite difficult to do efficiently.

Initially the implementation used each of these techniques separately. Each of the techniques had the advantages and disadvantages noted above, but neither technique was really satisfactory. The implementation now uses a mixed strategy: when the heap is first copied, the data is written to disk directly, taking advantage of the ability to do large block transfers and when the random writes are being transferred, RVM is used.

Other Issues

When a flip occurs, the from-space identifier must be updated to indicate which of the stable spaces is the current stable from-space. This update does not need to be made stable until the next `commit` after the flip. When the client first commits a transaction, that transaction will be applied to the new stable from-space. Sidney writes the from-space identifier via RVM, but using a "no-flush" transaction that does not require a synchronous disk write. Thus, although the client is paused while the garbage collector finalizes the flip and updates the roots, no synchronous disk write occurs during this interruption. If such a write were required, it would place a lower bound on the length of the GC pause that corresponded to the time to do a synchronous disk write.

Replication-based rollback also somewhat complicates the process of forwarding `writes` to their to-space replicas. When a transitory object that has a persistent replica is written, that `write` must be forwarded to the replica. Since the replica is persistent, the new value also must be recorded with RVM and potentially forwarded to volatile to-space as well. Both of these are accomplished by added the `write` to the persistent heap to the write log.

5.5 Some Final Implementation Details

It is important that some of the details of the implementation be very clear, since they play a significant role in the performance evaluation that follows. There are three key issues. First, how do the various threads synchronize as the collection proceeds? Second, what are the details of establishing the flip condition? Third, what are the steps needed to finalize the flip?

5.5.1 Synchronization Among Threads

When a collection is triggered, the collector obtains the current roots and begins copying the heap using the GC thread, while the client thread is allowed to resume executing user code. When the GC thread has copied a suitably sized block of data to to-space, it triggers the I/O thread to write this block to disk. Currently the block-size is 4096 bytes, but that value is not particularly significant. Initially, the I/O thread writes these blocks directly to disk. Eventually, the GC thread establishes the flip condition, but using the roots it started with initially, not the roots currently being used by the client. (Actually, if the client thread has entered the runtime since the GC thread obtained the roots, the client thread will update the roots, but nothing guarantees that this will happen.) The GC thread then waits for the I/O thread to finish transferring all of the current volatile to-space to disk. At this point, one *round* of collection is complete. The GC thread interrupts the client thread to obtain new roots and the current write log. After a parameterizable number of rounds (currently, two) are complete, the collector causes the I/O thread to change from doing direct writes to using RVM. The I/O thread periodically does an RVM commit, causing it to block. After another parameterizable number of rounds, again currently two, the GC thread causes the I/O thread to block. At this point, the GC thread takes over the responsibility for writing the to-space through RVM, which it does using non-blocking RVM commits. The reason for switching to using the GC thread is so that when the collector actually tries to flip, it will not be necessary to synchronize with the I/O thread. At this point, each time the GC thread interrupts the client thread, the client thread is allowed to do a bounded amount of work to reestablish the flip condition. If the client thread is unable to reestablish the flip condition in the allotted time, it unblocks the GC thread and returns to user execution. If it can be reestablished, then the client thread *finalizes* the collection, and the GC thread blocks waiting for a new collection. At this point, the I/O thread is unblocked, so that it can do the RVM log truncation. It sometimes happens that the I/O thread is still doing the truncation when a new collection begins. In this case, the I/O thread completes the truncation before starting to copy the to-space to disk.

5.5.2 Establishing the Flip Condition

What are the details of establishing the flip condition? The obvious requirements are that all of the data reachable from the persistent roots be copied to to-space and all of the write forwarding must be done. Pointers that point from the transitory heap to the persistent heap also must be used as roots. Currently, this is done rather crudely by examining each pointer in the transitory heap. Finally, the rollback log used for replication-based rollback contains pointers that must be used as roots, since they point to places in the persistent heap that are replicas of transitory heap objects and must be retained. When all of these actions have been done, and all of to-space has been written to disk or recorded with RVM, the flip condition is established.

5.5.3 Finalizing the Flip

When the client thread is able to establish the flip condition within its allotted time bound, it can then finalize the flip and terminate the garbage collection. The reason that the flip must be finalized is that when replicating collection uses a root to `copy` from, it does not update the root to point to to-space, because that would violate the from-space invariant. When the flip occurs, these roots must be redirected, or *forwarded*.

The following steps are required to finalize the flip. These steps will play an important role in the performance evaluation, since the length of the GC pause will turn out to depend on this part of the implementation more than any other. First, the collector forwards all the pointers in the transitory heap that point to the persistent heap, so that they point to to-space. This is done by looking at each pointer in the transitory heap. Next, the collector forwards all the pointers in the rollback log. Next, the collector forwards the roots. Next, the collector makes the volatile to-space be the volatile from-space, increments the heap sequence number, writes the new sequence number into the new volatile from-space, and records the `write` with RVM, effectively flipping the heaps in stable storage. No disk writes are needed; the new sequence number is forced to disk on the next user `commit`. Finally, the client thread unblocks the I/O thread so that it may do the RVM truncation, records the fact that the flip has taken place, and returns to executing user code. The GC thread remains blocked until a new collection is initiated.

Part II

Performance Evaluation

Chapter 6

Performance Overview

My thesis is that implicit persistent storage management techniques are superior to explicit techniques in terms of safety and competitive in terms of performance. Part I of the dissertation presents a detailed design and implementation of an implicit persistent storage management system, Sidney. In presenting the design of Sidney, I have established its safety advantages. I have also established that when it is possible to compare performance asymptotically, Sidney exhibits throughput that is comparable to implicit techniques. These asymptotic results are important because they show that Sidney is at no fundamental disadvantage when compared to explicit techniques. Unfortunately, asymptotic comparisons are not always possible, do not capture the issue of GC pauses, and may have no bearing on the performance seen on real applications. Thus, Part I does not serve to validate fully the performance claims of the thesis.

The goal of Part II is to complete the validation of the thesis by studying the performance of Sidney on a set of carefully chosen benchmarks. These benchmarks compare Sidney both to explicit techniques and to other implicit techniques. Sidney will have competitive performance if, barring issues not relating to performance, it can be substituted for another technique without making the resulting system's performance worse in a way that matters in practice. Of course, what matters "in practice" depends on the situation, but here I show that Sidney can be used in a wide variety of circumstances, without compromising performance.

To supplement the asymptotic analysis, I have measured Sidney's performance using four benchmarks: **trans**, **coda**, **comp**, and **OO1**. Taken together, these benchmarks allow me to investigate Sidney's performance in a wide variety of circumstances, from basic transaction processing to persistent programming environments. The results are favorable to the thesis; Sidney offers competitive performance in a wide variety of cases.

Two issues are central: throughput and latency. Throughput is important because if using Sidney slows down a system too much, it will not be used. This study shows that using Sidney may actually increase system throughput rather than reduce it. More critical is the question of latency; the main reason to introduce concurrent collection is to control GC pause times. If using GC creates pauses that are too long, the system's latency will not be competitive and again Sidney will not be used. This study shows that Sidney's pause times are not a function of the size of the persistent heap, are much shorter than the pauses created by stop-and-copy collection, and are somewhat shorter than the pauses created by the user when a `commit` is performed. The study also suggests ways in which the pauses could be shortened.

This chapter provides an overview of the performance evaluation. It starts with the high-level results of all the benchmarks. The details of the experimental methodology that are shared by all of the benchmarks follow, including a discussion of the statistical significance of the work. Detailed descriptions and results of each benchmark are presented in the following four chapters, one for each benchmark. Details concerning the accuracy of individual measurements are found only in these subsequent chapters. For the reader who is interested in only a high-level picture of the results, reading the first section of this chapter should suffice. Only readers who wish more details need to read the second section of this chapter or the detailed benchmark chapters that follow.

6.1 Overview of Results

This section is an overview of the performance results. It serves both to summarize the high-level results and to introduce the benchmarks and experimental techniques used in the rest of the evaluation. After some brief motivation and a summary of the results, the section divides into two parts. The first part is concerned with the throughput achieved by Sidney and with comparing it to the throughput of both explicit techniques and other implicit techniques. The second part focuses on the question of latency and compares the pauses created by Sidney to those created by stop-and-copy collection and `commit`.

The Challenge

If there were a suite of benchmarks that captured the important performance characteristics of typical applications using persistence and if that suite could be run with both an explicit persistence system and Sidney, then validating the thesis would be easy; I would simply run the suite and report the results. Unfortunately, the real world differs from this ideal one in several important ways. First, persistence is not a widely available technique and there is no consensus about what aspects of its performance are important. In fact, there are no widely used applications using general-purpose persistence. Second, in general, it is not possible even to run the same programs using Sidney and explicit persistence; the programming models differ too much and switching between GC and malloc-and-free is essentially impossible. Thus it is a challenge to validate the thesis.

Four benchmarks are used to meet this challenge: **trans**, a slightly modified version of the standard transaction processing benchmark, TPC-B; **coda**, a trace-driven benchmark that can measure either Sidney or a persistent malloc-and-free, using traces of the memory management behavior of the Coda file system; **comp**, the SML/NJ compiler modified so that its state is stored persistently; and **OO1**, a modified version of a standard object-oriented database benchmark.

These benchmarks were designed to answer the following questions:

- For transactions that modify only a small amount of data, can Sidney achieve the same throughput as an explicit system?
- How does system throughput differ when using Sidney, persistent stop-and-copy collection, or a persistent malloc-and-free?
- Can Sidney achieve speedups on real programs and are Sidney's GC pauses short enough to be non-disruptive?
- How does Sidney's performance change with the amount of live data in the persistent heap?

By using benchmarks from areas closely related to Sidney, the two standard benchmarks help address the problem of not knowing what performance characteristics are important for general-purpose persistence. These benchmarks are appropriate because Sidney is a transaction system that supports basic OODB functionality. **Trans** measures performance in the important special case when transactions modify and allocate only small amounts of data and when `commits` are frequent. The existence of a separate implementation of TPC-B based on explicit techniques makes it possible to use **trans** to compare the two approaches. **OO1** measures Sidney's performance on a simple engineering database and is used to study Sidney's performance when the livesize of the persistent heap varies. **Coda** allows a direct comparison to be made among concurrent GC, stop-and-copy GC, and malloc-and-free. The benchmark is based on a trace of the persistent memory management behavior of the Coda file system. The benchmark replays this trace, either calling exactly the same persistent malloc-and-free as used by Coda, or calling Sidney's allocation routines and allowing GC to reclaim storage. Finally, **comp** provides a benchmark that is based on a large preexisting program, rather than a program written primarily for benchmarking. **Comp** has long transactions that modify and allocate significant amounts of storage, making it quite different from the other benchmarks, in particular **trans**.

Summary of Results

The basic results support the thesis. First, consider throughput:

- On small transactions, both Sidney and the explicit version of **trans** are able to achieve close to the maximum possible throughput.
- When compared to persistent malloc-and-free, Sidney has lower overhead and achieves higher throughput.
- When compared to persistent stop-and-copy collection, Sidney is able to overlap GC work with user work and thus achieves higher throughput. Since the primary overlap is I/O with computation, higher throughput is achieved on uniprocessors, not just on multiprocessors.

Next, consider latency:

- The pauses created by Sidney's collector are at least an order of magnitude shorter than the pauses created by stop-and-copy collection.
- The pauses created by Sidney's collector are shorter and less frequent than the pauses created by `commit`.
- The length of pauses created by Sidney's collector are not a function the amount of live data in the persistent heap.
- In some cases, the pauses created by Sidney are still longer than ideal.

The subsections that follow substantiate these basic results.

6.1.1 Throughput

If the techniques used in Sidney make systems too slow, then despite their safety advantages they will not be used. This section examines the impact of Sidney on system throughput and shows that Sidney can offer improved system throughput in addition to greater safety.

Small-Transaction Throughput

Achieving good performance on small transactions is a central design goal for most transaction systems; RVM and Sidney are not exceptions. Since small-transaction throughput is so important, the Transaction Processing Council has created a standard series of benchmarks to measure small-transaction performance called TPC-A, TPC-B, and TPC-C. Since TPC-B is a well understood standard and is simple enough that it is possible to code both an implicit and explicit version, I used it as a basis for benchmarking Sidney's small-transaction performance.

Here, I present the results of running four programs, *trans-initial*, *trans-improved*, *trans-explicit*, and *trans-raw*; together these programs form the **trans** benchmark. The **trans** benchmark does not involve GC at all, but measures only the performance of `commit`. Trans-initial is a slightly modified version of TPC-B coded in SML and using Sidney. Trans-improved uses the same SML code as trans-initial, but uses a slightly improved version of Sidney. Trans-explicit is a similarly modified version of TPC-B coded in C++ that uses explicit techniques, static storage allocation, and RVM. Finally, since the speed that writes can be done synchronously to the disk limits the performance of these programs, I also present the results for a simple program, trans-raw that just does synchronous writes to the "raw" disk. Chapter 7 gives a complete discussion of the details of **trans**.

Figure 6.1 presents the throughput of **trans** on a DECstation 5000/200, a uniprocessor workstation. The y-axis is the number of transactions per second (TPS), a measure of the throughput of the system. The bars labeled Sidney (Initial) and Sidney (Improved) show the results for trans-initial and trans-improved, while the bar labeled Explicit shows the result for trans-explicit, and the bar labeled Raw shows the result for trans-raw.

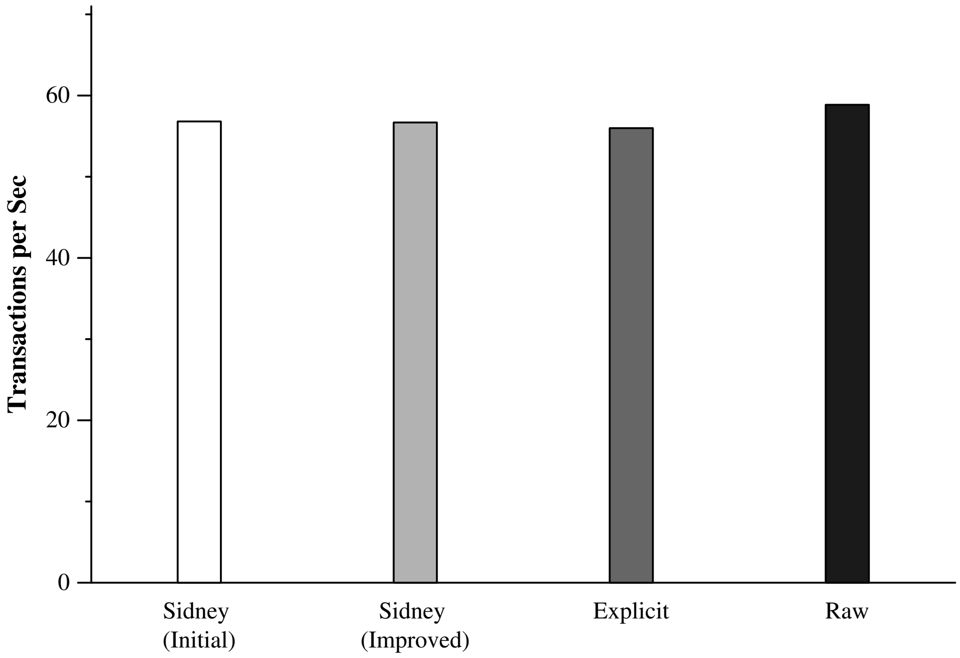


Figure 6.1: **Trans** Throughput (uniprocessor)

The disks used on the DECstation rotate at 3600 RPM. In the best case a single commit record can be appended to the log on each revolution, which would result in a rate of 60 TPS, very close to the rate achieved when writing the raw disk, 58.9 TPS. The explicit version and both versions of the benchmark based on Sidney achieve very close to the maximum possible TPS. This result shows that using Sidney does not compromise the small-transaction throughput in this case and that in fact it is limited by disk performance.

Figure 6.2 presents the throughput of **trans** on a SGI 3D/40, a four-processor machine. The axes and labels are the same as for Figure 6.1.

The disks on the SGI rotate at 5400 RPM, allowing a theoretical maximum of 90 TPS. Trans-raw, trans-explicit, and trans-improved all achieve close to this rate. However, trans-initial shows greatly reduced throughput, achieving only 54 TPS. This performance is less than the 57 TPS trans-initial achieved on the slightly slower uniprocessor with slower disks. Investigation showed that the computation is expensive enough that the log write often misses a revolution of the disk. Reducing the computational overhead by a small amount resulted in trans-improved, which shows no performance degradation relative to the other benchmarks. These issues are considered in detail in Chapter 7.

Garbage Collection Versus Malloc-and-Free

How does the performance of GC compare to malloc-and-free? This is a key question, but it is difficult to answer because programs are written for a particular technique, and only in relatively special circumstances [63] is it possible to switch between them. It is even more difficult when dealing with persistence, since there are very few applications that use persistence in any form.

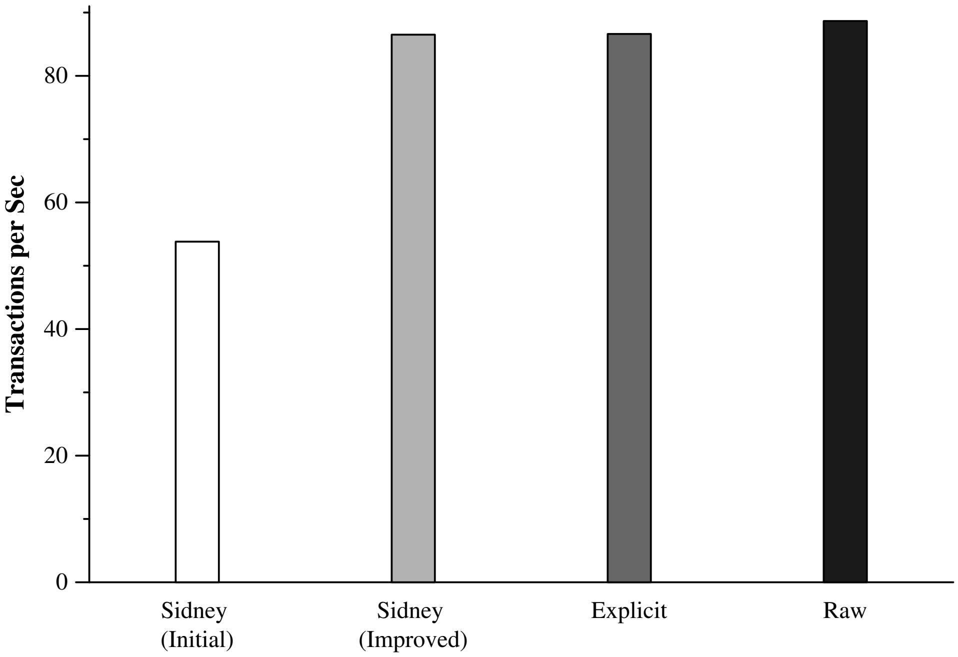


Figure 6.2: **Trans** Throughput (multiprocessor)

I solved this problem by factoring it into two pieces, first tracing a significant application's use of a persistent malloc-and-free to capture realistic program behavior, and then replaying the trace using a simple test program designed to support replaying the traces using either malloc-and-free or Sidney. I chose to trace the Coda distributed file system for several reasons. First, it is a large systems application in production use that uses general-purpose persistence. Second, it uses an explicit persistence system also based on RVM, facilitating the comparison to Sidney. Finally, I had access to the system and its designers, facilitating the addition of tracing code and understanding the results. The trace replay program is very simple; basically it can either redo exactly the same persistent `mallocs` and `frees` done by Coda, or use Sidney to do allocation and GC, "redoing" the frees by making the freed storage unreachable. I call this the **coda** benchmark. A more detailed analysis of **coda** can be found in Chapter 8.

The trace used in the measurements comes from tracing the Coda client program on my workstation for almost a month. During this time, I used Coda extensively for text preparation, program development, and analyzing and storing benchmark results; typical uses of Coda. Even in the worst case, the trace takes only a little over a minute to be replayed. This short time to replay the trace implies that persistent storage management is not an important factor in Coda's throughput, and that the question of whether to use implicit or explicit storage management in Coda must be answered on grounds other than throughput.

Figure 6.3 shows the performance of Sidney on the uniprocessor when replaying a trace representing the allocation of a little more than 1 MB of persistent data. The x-axis is the number of GC flips achieved by Sidney. The y-axis is the time to replay the trace in

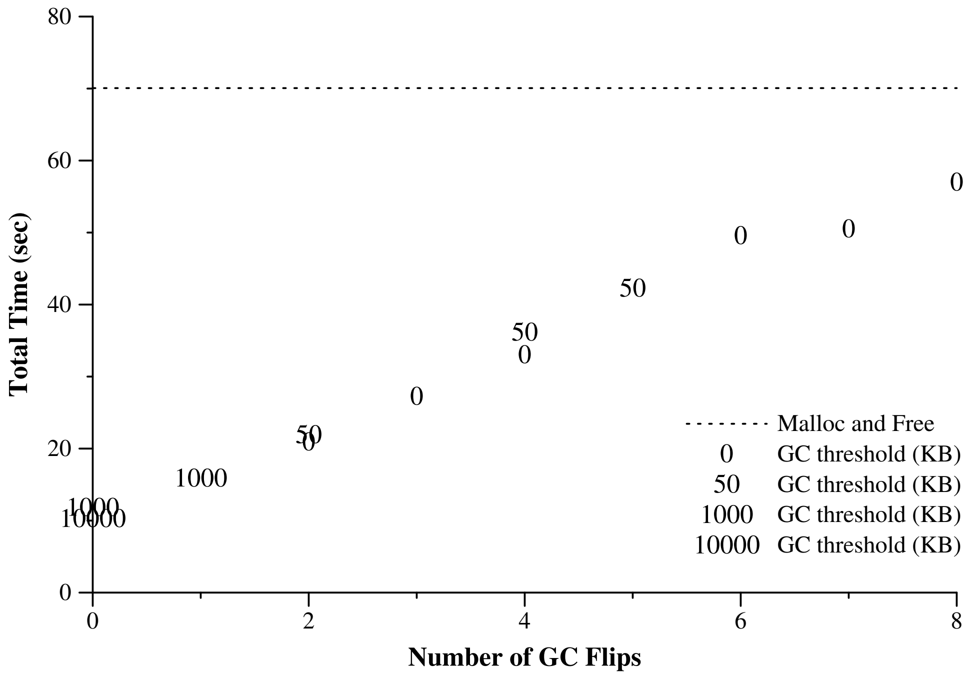


Figure 6.3: **Coda** Performance (uniprocessor)

seconds. The numbers plotted represent the setting of the GC threshold parameter (for the persistent heap) in kilobytes. The time to replay the trace using malloc-and-free is shown as a dotted trace line near the top of the plot.

The GC threshold controls how much allocation is done after a flip and before another collection is initiated. A value of 0 KB means that immediately after each collection completes, a new one is started. Since the trace represents the allocation of just over 1 MB of persistent data, a value for the threshold of 10000 KB guarantees that the collector is never started. Since the collector is non-deterministic, any particular value of the threshold can lead to the collector flipping different numbers of times. The plot shows that the throughput is mostly a function of the number of flips, rather than the threshold parameter. Even in the most expensive case, when the collector flips 8 times or about every 125 KB of allocation, GC is still less expensive than malloc-and-free. When the collector flips only once, it is 3.5 times faster than malloc-and-free. In practice, on these platforms, GC thresholds are usually set to several megabytes. Chapter 8 shows why Sidney is able to offer improved performance, and also examines the performance of stop-and-copy collection using the trace.

Concurrent Versus Stop-and-Copy Collection

Although the focus of the thesis is on showing that Sidney is competitive with explicit techniques, it is also useful and important to compare Sidney to other implicit techniques, and in particular to stop-and-copy collection. This allows us to explore what performance improvements might be achieved by using Sidney in systems that already use implicit

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Total Time		Concurrent	70.4	100.0	-0.3	0.5
Total Time	CPU	Concurrent	62.8	100.0	-0.2	0.2
Total Time		Stop&Copy	76.5	100.0	-0.4	0.3
Total Time	CPU	Stop&Copy	62.9	100.0	-0.2	0.2

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Total Time	-6.1	-0.2	7.6	14.0

Table 6.4: **Comp** Times (uniprocessor)

techniques. This comparison is easy to perform because Sidney's implementation also includes a stop-and-copy version of the collector.

One benchmark that is useful for this comparison is a version of the SML/NJ compiler that has been modified to make its data structures persistent. A `commit` occurs after each file is compiled. The benchmark itself uses this modified compiler to compile a number of files (37) from the SML/NJ distribution. I refer to this benchmark as **comp**; a more detailed analysis is found in Chapter 9.

Table 6.4 shows the times for **comp** on the DEC uniprocessor, and includes both the results for concurrent and stop-and-copy collection. The top table is the primary data and has the following format:

Quantity Measured The quantity being measured. If the measurement is for CPU time, it is noted explicitly; otherwise, the measurement is for elapsed time.

Collector Used The collector used for the measurement, stop-and-copy or concurrent.

Median The median value of the quantity being measured in seconds.

Total The percentage of the total time the median represents. This percentage is based on the total time for the same collector and for the same choice of elapsed or CPU time. (Since we only present total times in this table, this column is always 100%. In later measurements of individual parts of the system, this information will be more useful.)

25th The value of the 25th percentile as a percentage relative to the median.

75th The value of the 75th percentile as a percentage relative to the median.

The bottom table contains the differences or deltas between various medians in the primary data and has the following format:

Delta The quantity the delta is for.

Elapsed The difference between the elapsed times for the stop-and-copy and the concurrent collectors in seconds.

CPU The difference between the CPU times for the stop-and-copy and the concurrent collectors in seconds.

Concurrent The difference between the elapsed and CPU times for the concurrent collector in seconds.

Stop&Copy The difference between the elapsed and CPU times for the stop-and-copy collector in seconds.

Both of these formats are standard and used throughout the rest of the dissertation.

Using concurrent collection results in a speedup of 6.1 sec, or 8%, compared to stop-and-copy collection. Since the machine is a uniprocessor, this speedup comes entirely from overlapping I/O with computation. The multiprocessor shows similar results, although it also benefits from using an independent processor for the GC thread. The speedups are modest, but Chapter 9 shows that these speedups are a significant fraction of the maximum possible, given the fraction of total time spent in GC.

Dependence on Livesize

The livesize of the heap is a fundamental factor in the cost of collection. To investigate this issue, I used the **OO1** benchmark to measure the behavior of both Sidney and stop-and-copy collection as the livesize in the heap increases.

OO1 is an industry-standard, object-oriented database benchmark. It models a simple database of factory parts. The standard version traverses and modifies the data structures representing the database, as well as adding new parts. Since this computation does not create any garbage, it is not an ideal test case for a garbage collector. To create the **OO1** benchmark, I modified OO1 so that it also deletes the same number of parts as were added, keeping the amount of live data in the heap constant. Since the livesize to run the benchmark is constant, it is easy to study the dependence on livesize by just adding controlled amounts of extra live data to the persistent heap. A more detailed analysis of **OO1** can be found in Chapter 10.

Figure 6.5 shows the results of running the **OO1** benchmark on the uniprocessor, which performed 50 iterations of the basic benchmark. The y-axis shows the average elapsed time in seconds for one iteration of the benchmark, and the x-axis is the size of the heap in megabytes. The minimum heap size (4.5 MB) is dictated by the amount of data needed to run the benchmark, while the maximum livesize (21 MB) was chosen to match the onset of paging on the uniprocessor.

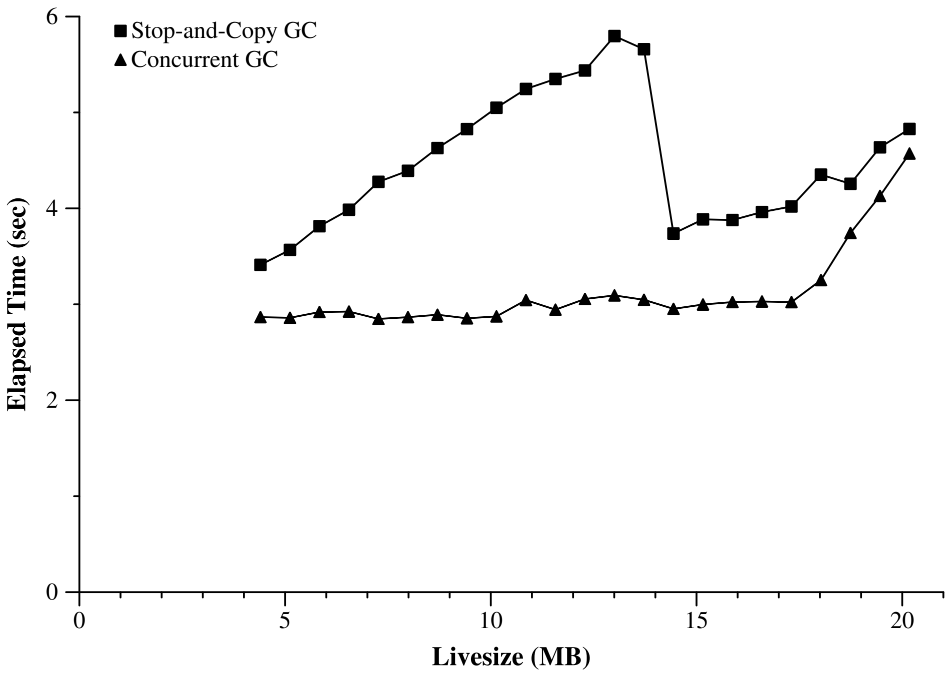


Figure 6.5: **OO1** Throughput (uniprocessor)

The elapsed time for the concurrent collector is constant except at the largest livesizes, and it performs uniformly better than the stop-and-copy collector. The execution time is constant because the collector is configured so that it is always running. Although this is not how the collector would be configured in practice, an aggressive setting emphasizes the overheads of collection and makes it easier to measure the overhead of collection. Since the work expended by the collector is approximately constant per unit time, the number of flips that it can achieve decreases as the livesize increases.

The elapsed time for the stop-and-copy collector shows a surprising saw-toothed shape. The sawtooth comes from a combination of effects. The linear increase of the elapsed time with the size of the heap comes about because the cost of each collection increases linearly with the livesize. The sudden drop in the elapsed time is an artifact of the benchmark's setup. If the amount of data copied by the concurrent collector differs from the amount copied by the stop-and-copy collector, the comparison of the two techniques will be biased. To make the amount of data copied the same, it is necessary to force the two collectors to flip at the same points relative to the program's execution. How this is accomplished is discussed in detail in Subsection 6.2.1. The drop occurs when the number of flips achieved by the concurrent collector changes, in this case from two to one. The slope of the increasing line should be $c * N$, where c is a constant and N is the number of flips. Thus the ratio of the slopes of the two parts of the sawtooth should be 2:1, which is approximately true.

The final puzzle is "Why does the concurrent collector curve go up suddenly at the largest livesizes?" At these sizes, the working set size of Sidney is approaching the size of the available physical memory. When this limit is reached, the program begins to page out and thus slows down. Since the concurrent collector must keep both the working set of the client and of the collector in memory at the same time, it has a larger working set, and thus begins to page sooner. Informal observation of the paging rate confirms this explanation.

6.1.2 Latencies

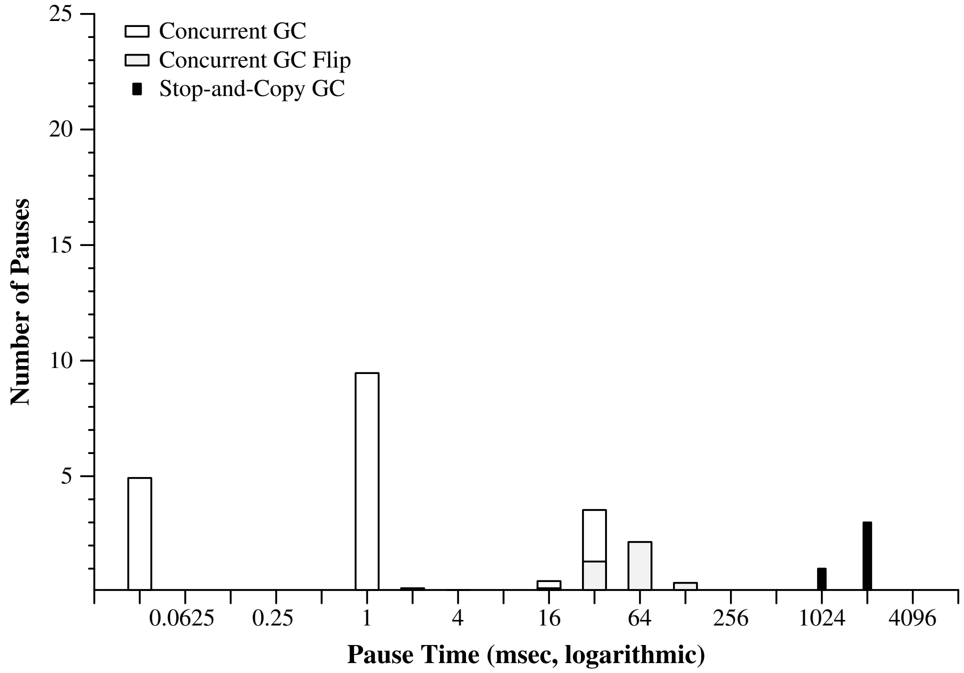
The measurements above demonstrate that Sidney and concurrent garbage collection provide good throughput relative to both explicit techniques and other implicit techniques. However, the primary goal of adding concurrent collection is to limit pause times, not to speed up programs. In this subsection, I look at pauses caused by Sidney. The key questions are "How long are the pauses created by the collector?" and "Would they be disruptive to the user?" Only **comp** and **OO1** are discussed here. **Trans** does not involve collection, and its `commit` pauses are discussed in Chapter 7. The GC pauses for **coda** are artificially low due to the nature of the replay program; they are not discussed in the dissertation.

Detailed Latencies

Comp is the only one of the benchmarks used in the dissertation that was not designed exclusively for use in performance evaluation. Thus, the results derived from it are more likely to be representative of real programs than the other benchmarks. For this reason, I chose **comp** as the primary benchmark to study pause times in detail. I compare the lengths and frequency of the pauses due to concurrent GC to those due to two other sources: stop-and-copy collection and `commit`. If concurrent collection cannot achieve pause times significantly shorter than stop-and-copy collection, it will clearly be a failure. On the other hand, if the pauses created by concurrent collection are both shorter and less frequent than those created by the user doing `commit`, then Sidney's pauses will not be disruptive and Sidney will have competitive performance.

Figure 6.6 shows the results of the pause time measurements for the compiler benchmark on the SGI multiprocessor. The top graph shows the garbage collector pauses and the bottom graph shows the `commit` pauses. The y-axis is the number of pauses. The x-axis is the length of the pauses in milliseconds; it has a logarithmic scale, although the hash mark labels are the actual pause times, not their logarithms. The pauses themselves have been placed in bins such that all pauses greater than 2^N and less than or equal to 2^{N+1} are placed in the $(N+1)th$ bin. Each hash mark label also denotes the maximum pause time for a given bin. The scales of each plot is the same to facilitate comparison. In the upper plot, the stop-and-copy pauses are black; the pauses of the concurrent collector that do not result in flips are white, and the pauses that result in flips in gray. In the lower plot, all of the pauses are for `commit`; the pauses that result from commits when the system uses stop-and-copy collection are shown in black, and those that result from commits when the system uses concurrent collection are shown in gray.

The top plot shows that the stop-and-copy pauses are long, 1 to 2 sec, and easily noticeable by a human user. In contrast, the pauses caused by the concurrent collector are much more numerous, but are also at least an order of magnitude shorter. The longest pauses that do not result in flipping are 32 msec long, while the pauses for flips are almost all are bounded by 64 msec, with only a small fraction between 64 and 128 msec. 50 msec



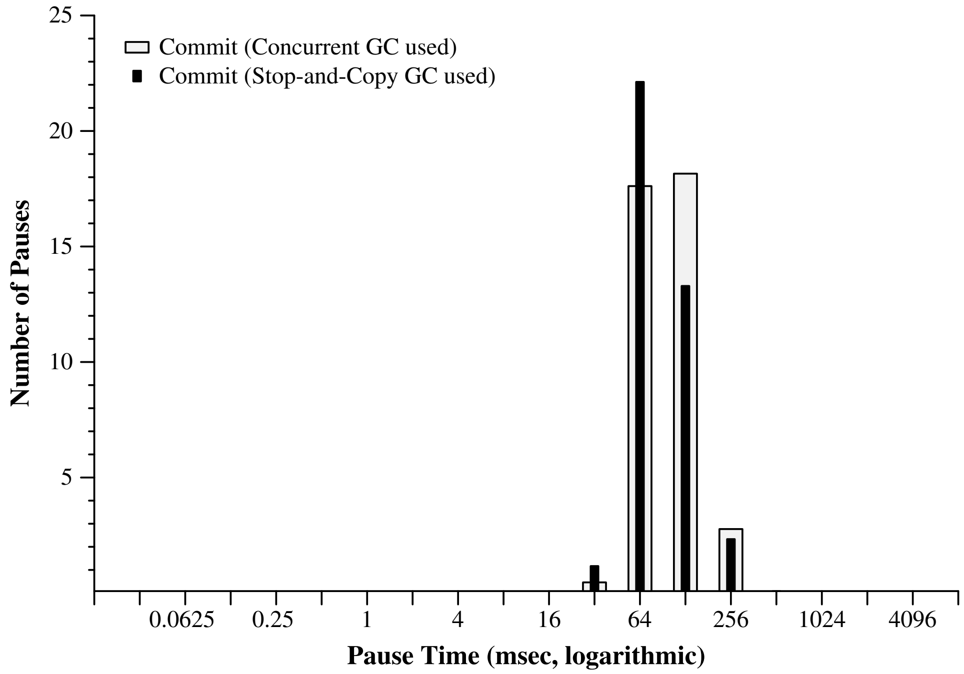


Figure 6.6: **Comp** Pause Times (multiprocessor)

is generally considered the threshold of human perception [10]. The shortest concurrent GC pauses arise because of the need to synchronize the client thread with the collector. These pauses are frequent, but brief enough to be non-disruptive. The next group of non-flip pauses, at around 32 msec, result when the collector, running as part of the client thread, reaches the limit of the number of bytes it is allowed to copy and returns control to the user code. The length of these pauses can be controlled by changing the copy limit parameter. The flip pauses occur when the collector actually completes a collection and flips. Even though there are more concurrent GC pauses, in Chapter 9 we shall see that the total time spent in GC is much less for concurrent collection than for stop-and-copy collection.

Now consider the `commit` pauses in the lower plot. The `commit` pauses created when concurrent collection is in use are somewhat longer than those created when stop-and-copy collection is in use. This difference is because concurrent GC interferes with the client thread slightly. Compare the `commit` pauses to the GC pauses in the upper plot. In general, `commit` pauses are shorter in length than the stop-and-copy pauses, but the same length or longer than the concurrent pauses. There are many more `commit` pauses than GC pauses, particularly if the very short concurrent GC pauses are excluded. That the `commit` pauses, which are caused by the user, are both longer and more numerous than the GC pauses means that the GC pauses will tend not to be disruptive.

Dependence on Livesize

One of the significant disadvantages of stop-and-copy collection is that the pauses created by collection increase in length with increasing livesize. Replicating concurrent collection does not suffer from this problem. In this measurement, **OO1** is used to study the dependence of GC pause times on livesize. No comparison of the `commit` pauses is made here, such a comparison is found in Subsection 10.2.3.

Figure 6.7 shows two views of the pause times due to stop-and-copy collection and concurrent collection on the uniprocessor. The x-axis is the livesize in megabytes. The y-axis is the length of the pause in seconds. The two plots have radically different y-axis scales. The top plot includes the results for both stop-and-copy collection and concurrent collection; the bottom plot omits the stop-and-copy information so that it may have a much larger and more useful scale.

The top plot shows that the length of the stop-and-copy pauses shows a strong dependence on the livesize, increasing by about a factor of three as the heap grows by 16 MB. In contrast, on this scale, the concurrent pauses not only are constant as livesize varies, but are also almost zero. The lower plot shows just the concurrent pauses on a much larger scale. Even at this scale, the non-flip pauses are of almost zero length and do not depend on the livesize. The short length of these pauses is because the typical non-flip pause is caused by the need to very briefly synchronize the client with the concurrent collector. The concurrent flip pauses are much longer than the non-flip pauses; although these pauses are not "constant," they do not show any systematic dependence on the livesize. Unfortunately, the lengths of these pauses, around 1 second, are still long enough to disrupt the user. Chapter 10 examines the cause of these long pauses and proposes some solutions.

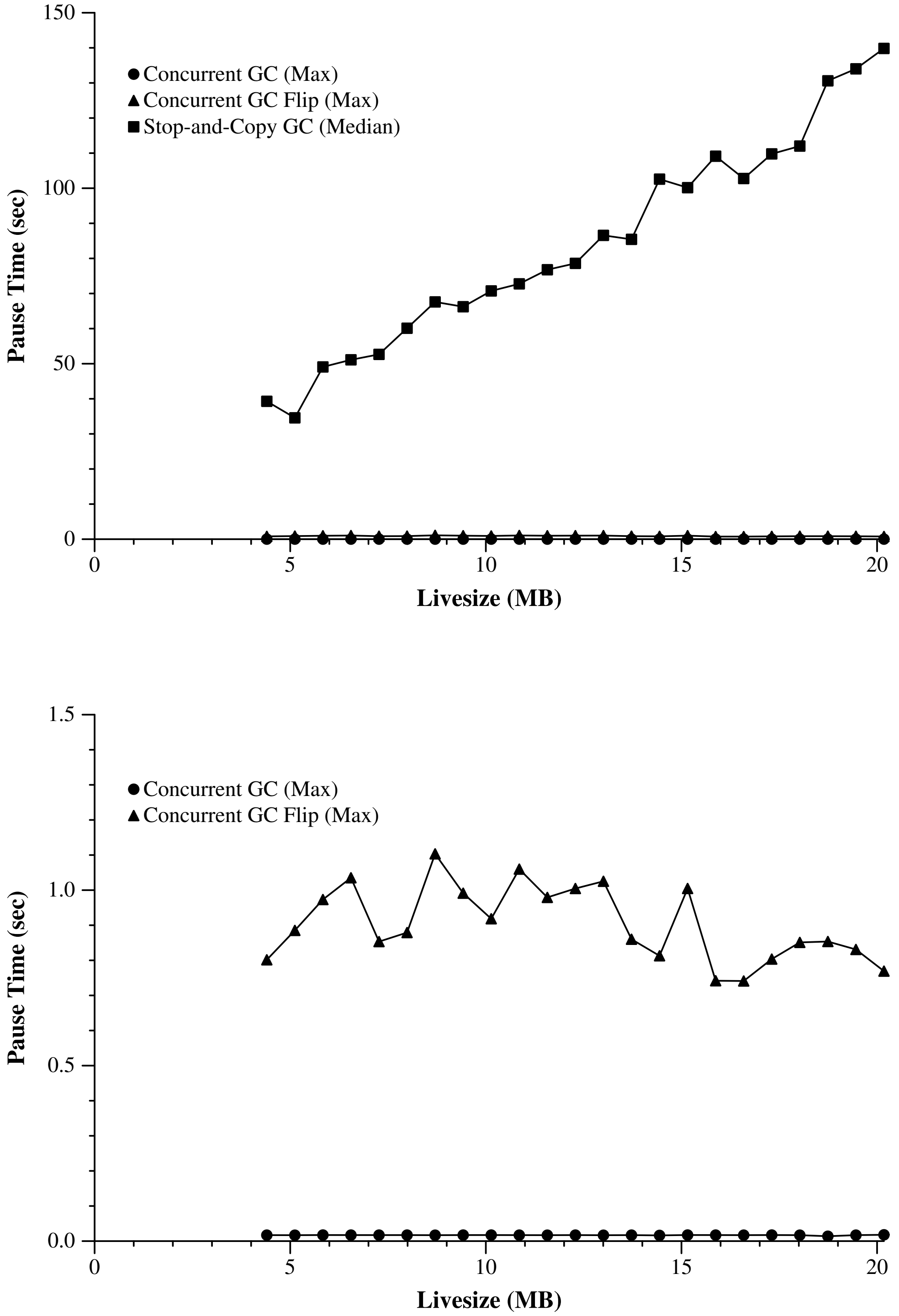


Figure 6.7: **OO1** GC Pause Times (uniprocessor)

6.2 Experimental Methodology

In this section, I discuss a number of details about the experimental setup, data analysis and presentation. This section covers only issues that are common to all the benchmarks; details specific to an individual benchmark are found in their corresponding chapter. This section has three parts. The first explains the details of the experimental setup, such as what machines were used. The second section covers the statistical issues that arose during the study. Finally, I also discuss issues in the presentation of data.

6.2.1 Experimental Setup

The uniprocessor used was a DECstation 5000/200 with a 25 MHz MIPS R3000 CPU. This machine has 128 MB of physical memory and runs the Mach 2.6 operating system [7]. The system has separate instruction and data caches of 64 KB each. The minimum latency of a synchronous disk write is 16.66 msec and the maximum disk bandwidth is 400 KB per second. The system can copy and scan live data at a rate of about 1 MB per second.

The multiprocessor used was a Silicon Graphics 4D/340 with four 33 MHz MIPS R3000 CPUs. This machine has 256 MB of physical memory and runs IRIX version 4. Each processor has a 64 KB instruction cache, a 64 KB primary data cache, and a 256 KB secondary data cache. The secondary data caches are kept consistent via a shared memory bus-watching protocol and there is a five-word deep store buffer between the primary and the secondary caches. With this configuration, the system can copy live data at a rate of between 1 and 2 MB per second. The minimum latency of a synchronous disk write is 11.11 msec and the maximum disk bandwidth is 1600 KB per second. The disk bandwidth for this machine is a factor of four greater than for the uniprocessor. I suspect this is due to a file-system that is better optimized for large data transfers.

On both systems, three disks are used. One disk holds the system and user file systems. The second disk holds the two data segments used as the stable representation of persistent from-space and persistent to-space. Measurements showed that the best performance resulted when these segments were stored and manipulated as Unix files. The final disk holds the RVM log. Manipulating the log is most efficient when it is used as a “raw” device with no file system. Using three disks prevents disk arm contention from complicating the measurements. In particular, since the disk holding the RVM log is used for no other purpose, seeks are rare when writing the log.

The uniprocessor was dedicated to benchmarking and no other users were active during benchmarking, although the standard system daemons were active. The multiprocessor was not so dedicated; however, it was lightly loaded and there were adequate CPU and memory resources, so that users did not compete with the benchmarks for resources. There is no evidence that interference was any greater of a problem on the multiprocessor than on the uniprocessor. Unfortunately, it was not feasible to isolate either machine from the network. Thus, it is plausible to believe that the measurements occasionally reflect some interference from the network, especially on the uniprocessor, which must multiplex its single processor. Interference from other programs will tend to create outliers in the measurements; how such outliers are handled is discussed in Subsection 6.2.2.

Clocks

The DECstation 5000/200 is equipped with a special timing board to provide better clock resolution than the standard system clock. This clock measures elapsed time in increments of 25 nanoseconds. The overhead to access the clock is about 200 usec, so in practice the effective resolution is 200 usec. The CPU clock does not use the special board and has a resolution of 16.6 msec.

The SGI 4D/340 has an elapsed time clock with a resolution of 800 usecs. The CPU clock has a resolution of 10 msec and the overhead for reading the clock is 200 usec.

In general, I only measured one aspect of the system at a time. This means that individual timer calls are kept to a minimum and do not interfere with each other. On both machines, CPU times are measured on a per thread basis. In later chapters where CPU times are presented, the reader should be careful to keep this in mind.

A final point is that most of the measurements reported here are the result of averaging a number of timer calls. This process will tend to increase the accuracy of these measurements, although I have not attempted to quantify this effect. The only measurements that typically involve only one reading of the clock are the pause time measurements.

Flip Tracking

The total amount of data copied is the primary component of the cost of GC. Setting the GC threshold parameter to the same value for concurrent and stop-and-copy collection will result in the collectors flipping at different points and even flipping vastly differing numbers of times. To do a controlled comparison between the two requires that they flip at the same points, so that they copy the same amounts of data. (This statement is not strictly true, because concurrent collection may copy some data that becomes garbage. Since this extra copying is an overhead of concurrent GC, it is proper to include this copying.)

It would not be too hard to make the two collectors flip at the same points, except that the concurrent collector is non-deterministic and thus does not even consistently flip at the same points itself. I overcame this problem by having the concurrent collector produce a trace of its flip points and then replaying that trace using the stop-and-copy collector, forcing it to flip at the same points.

I tested the trace generation and replay code to see if it biased my results for or against one of the two techniques. I did this by running the concurrent collector with tracing on and off and running the stop-and-copy collector using a trace that produced flips at the same points as using the GC parameters. In both cases using the trace caused an increase in execution times well below 1%. Furthermore, the cost of producing a trace and consuming one was well matched, further reducing the bias.

GC Parameters

One final issue concerns how the various GC parameters were set during the experiments. Some of the parameters, notably the thresholds for triggering major, minor, and persistent collections have a great deal of influence and accordingly are set individually for each benchmark. These settings are discussed in the chapter for the specific test. However, many parameters are of less importance and were set in the same way for all tests. These are described here. The minor parameters, settings, and reasons for them are:

In-line Copy Limit

This parameter (L in Table 5.4) controls the maximum amount of copying that can be done when the GC is executing in the client thread and trying to flip. This parameter is set at 10 KB, because smaller values did not result in lower maximum pause times.

Concurrent Copy Limit

When the collector is active in the GC thread, it polls to see if it needs to relinquish control to the client thread. The parameter (U in Table 5.4) controls how much copying is done between polls. It is set at 10 KB, because smaller values did not result in any noticeable changes in performance while larger values impacted pause times.

Further, note that although the collector for the transitory heap can also be run concurrently, this was not done in any of these measurements.

6.2.2 Statistical Issues

To analyze the data generated in this study, I wanted three statistical measures. First, a measure of the central tendency of the distribution being sampled, for example, the mean. Second, a measure of the width of the distribution, for example, the standard deviation. Third, a measure of the confidence in the results as a function of the number of samples, for example, the t-test. These measures must be well justified statistically and robust given the nature of the distribution of values encountered in practice.

The Problem

The basic statistical difficulty I face is graphically illustrated by Figure 6.8. This plot shows the distribution of total elapsed time for five hundred runs of trans-improved on the uniprocessor. The x-axis is the elapsed time in milliseconds. The y-axis is the probability with which a particular elapsed time occurs. The boxes and dotted line represent the experimental measurements. The measurements have been placed in bins such that the width of the bins is a little less than 0.1% of the values being placed in bins. The solid line is a normal distribution with the same mean and standard deviation as the data; it has been scaled to be the same height as the mode of the data. The mean, median, 25th, and 75th percentiles (or quartiles) are shown on the x-axis. Note that the median is denoted by a "m" and is a bit hard to see.

This distribution is typical of those found in this study. The problem with this distribution is that it deviates substantially from a normal distribution. First, the distribution is skewed: much of the normal curve is chopped off on the left. Second, the distribution has a long tail: there is a significant probability density long after the normal distribution has dropped to

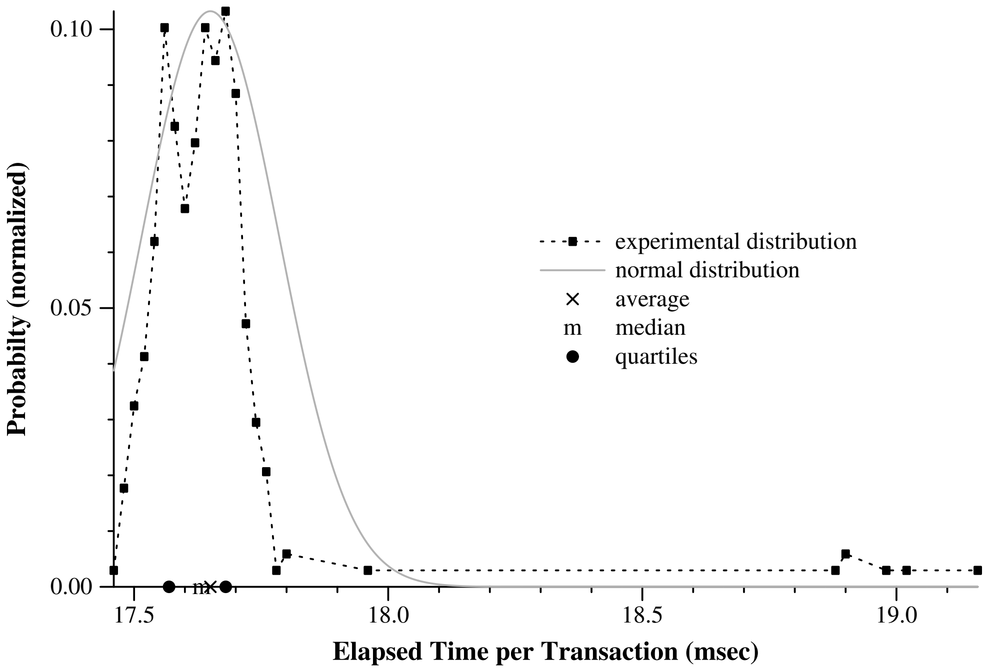


Figure 6.8: Distribution of **Trans** Elapsed Times (uniprocessor)

zero. These deviations from normal are important because they make the typical statistical measures mentioned above less desirable to use. The calculation of means is sensitive to the presence of long tails. The standard deviation is also sensitive to the long tail. This sensitivity is seen in the figure by noticing that the normal curve is much broader than the bulk of the experimental data. Also the standard deviation captures no information about the skew of the data. Finally the standard tests used to establish confidence intervals depend on a normal (or t for small sample sizes) distribution of the data.

The Solution

Since the usual measures are not well suited to the data at hand, I have investigated and used alternative techniques. One technique I rejected was to truncate or chop off the tails of the distributions and then to apply the standard measures. Although a common practice, I rejected it because I could not find good statistical justification for doing so, and in particular for how to justify how much of the tails to truncate.

Following the suggestion of Jain [29] as well as other authors [32, 45], I report the median as a measure of the central tendency of the distribution. The median is both well understood from a statistical point of view and quite insensitive to outliers. It is also easy to compute and understand.

Also following Jain [29], as a measure of the width of the distribution I report 25th and 75th percentiles. Again, these measures are easy to compute and understand. Unlike the standard deviation, they also give an indication of the skew of the distribution.

The final question is what measure of confidence to use. Pragmatically, the question I wanted to answer is how many runs of each measurement did I need to achieve a given level

N	Expectation	N	Expectation
10	10.9%	20	1.2%
11	6.5%	21	2.7%
12	3.9%	22	1.7%
13	9.2%	23	1.1%
14	5.7%	24	0.7%
15	3.5%	25	1.5%
16	2.1%	26	0.9%
17	4.9%	27	0.6%
18	3.1%	28	0.4%
19	1.9%	29	0.8%

Table 6.9: Expectation That the Median Lies Outside the Reported Quartiles

of confidence in the median. Using a technique based on non-parametric statistics outlined below, I found that by using 24 repetitions of each measurement, I can have a confidence of 99% that the actual median lies between the measured quartiles. This approach has the advantage that the quartiles now do double duty, both giving an idea of the shape of the distribution and bounding the error of the median.

The Justification

Reporting medians and quartiles instead of means and standard deviations is common statistical practice and requires no additional justification. Unfortunately, these measures do not indicate how statistically significant the results are. In particular they give no guidance about how many samples are needed to determine these quantities to a given level of accuracy. Since it is not feasible to repeat all measurements 500 times, it is important to gain an understanding about how many repetitions are needed in practice.

Fortunately, there exist non-parametric methods (that is methods that make no assumption about the underlying distribution) to determine confidence intervals as well. I discuss only the simplest technique here, it is based on a technique called the sign test; for information on more sophisticated methods see [45].

The goal is to find the probability that the median lies outside the sample quartiles as a function of the number of samples. If we sample a distribution, the chances that the sample value is greater than the median is, by definition, 50%. Now, consider a sample of size N. The chance that all N values are above the median is 0.5^N. In general, the chance that M of the N values will fall above the median is given by the binomial distribution for a probability of 50% success. Now we can find the probability that the median falls below the first quartile by summing the probabilities from the binomial distribution from 0, which is the probability that all the samples lie above the median, to $N/4$, which is the probability that one-quarter of the samples lie below the median. The probability that the median lies outside both quartiles, not just below the first quartile is just twice this sum. Table 6.9 gives this probability for some values of N. The expectation does not decrease monotonically with N because of the discreteness of the binomial distribution.

From the table we see that at around 24 samples, the probability that the median lies outside of the reported quartiles approaches 1%. For this reason, I choose 24 as the minimum number of samples for all the results reported here. These bounds are not meant to be particularly tight, and in fact the confidence we can have in the medians is likely to be greater than suggested here. However these bounds are tight enough for my purposes.

6.2.3 Data Presentation

There are two points of the data presentation that deserve comment, both having to do with pause times. The basic problem arises because the range of pause times is very large. The most frequent pauses are short; the most disruptive pauses are long but infrequent.

It is easiest to get a sense for the pause times created by the collector by looking at the actual distribution of pause times, rather than some summary like the median pause. Since the pauses can range in length from microseconds to tens of seconds, using a linear scale is difficult. Instead, unless the range of pause times is small, I present the pause time distributions with logarithmic scales, allowing a very wide range to be accommodated. This presentation obscures some fine detail, but is acceptable since it still captures the basic distinctions that are of importance.

A somewhat tricker problem arises when it is impossible to present the full distribution. An example of this is when presenting the pause time dependence on livesize. Now the question of how best to summarize the distribution arises. The problem is that there are many short non-disruptive pauses and a few long disruptive ones. In this case, the mean or median pause time is almost entirely a function of the short pauses and yet, it is the long pauses that are most important to the user. On the other hand, the maximum pause time is especially sensitive to interference from a variety of sources of error and will increase monotonically as more samples are taken. A metric that gets worse just because more samples are taken seems undesirable. Instead, I use the following metric. For each run, I find the maximum pause time, which is the one that is most disruptive and thus the best measure of how bad the pauses are for the user. Then across all runs I find the median of the maximums. This approach has the same basic advantages as taking the median for any value: it is a good measure of the central tendency and is insensitive to outliers. Furthermore, unlike the maximum observed value, the median of the maximums does not increase monotonically with a larger sample size.

I did not attempt to present any information about when GC pauses occur relative to other GC pauses. Some GC techniques lead to clusters of pauses just after a flip. This is not a problem with replicating collection. With Sidney, we expect to see a few short pauses to sample the roots, followed by a flip pause.

6.3 Summary

This chapter is an overview of the results of evaluating Sidney's performance. The goal of the performance evaluation is to show that in a wide set of circumstances, the storage management approaches used in Sidney result in performance comparable to that of explicit techniques. A primary difficulty in demonstrating this point is the lack of standard benchmarks and the difficulty of making direct comparison of Sidney to explicit techniques. To overcome this difficulty, I used four benchmarks, **trans**, **coda**, **comp**, and **OO1**. Two of these, **trans** and **OO1**, are standard benchmarks from closely related areas. The other two, **coda** and **comp**, are based on real programs, the Coda file system and the SML/NJ compiler respectively. **Trans** and **coda** allow a comparison to be made between implicit and explicit persistent storage management, while **comp** and **OO1** allow a detailed comparison of concurrent to stop-and-copy collection.

The results of running these benchmarks show that Sidney has good performance in a wide variety of circumstances and in general provide strong support for the thesis. For small transactions, the **trans** benchmark shows that Sidney and explicit techniques provide similar performance and are both limited by disk performance. Comparing concurrent collection to a persistent malloc-and-free showed that even a system that performs collections very aggressively can have lower run times than explicit techniques. For **comp** the speedups achieved by using concurrent collection are modest, but more importantly, using concurrent collection drastically reduces the length of GC pauses, yielding a system in which GC pauses are shorter and less frequent than the `commit` pauses. **OO1** showed greater speedups than **comp** because it has more GC overhead to optimize away. More importantly, **OO1** showed that performance of concurrent collection is basically constant as the livesize of the persistent heap increases. The length of GC pauses for concurrent collection and **OO1** are also independent of livesize. The following chapters explore these results in much greater detail.

Chapter 7

The Trans Benchmark

The designs of both Sidney and RVM pay special attention to achieving good performance on transactions that modify only a small amount of data. The **trans** benchmark is designed to investigate Sidney's performance on such small transactions using an industry standard benchmark. A further goal is to compare Sidney to an explicit persistence system based on RVM using the same benchmark. A final goal is to establish an upper bound on small-transaction throughput and to compare both Sidney and explicit persistence to this bound.

Most database and transaction systems place considerable importance on the performance of small transactions as do their customers. The need to quantify small-transaction performance has led the Transaction Processing Council [55] to design a number of benchmarks for this purpose. **Trans** uses a variant of one of these benchmarks, TPC-B, to measure the small-transaction performance of Sidney. **Trans** also uses a similar variant of TPC-B to measure the performance of an explicit persistence system, also based on RVM. Finally, **trans** also measures the rate at which synchronous writes can be done to the disks used for the RVM log. This rate establishes an upper bound on the possible performance of the TPC-B variants, whatever persistence technique is used.

The results of **trans** provide strong support for the thesis, as was seen in Chapter 6. On the uniprocessor, both Sidney and the explicit version achieve close to maximum possible performance. On the multiprocessor, the explicit version achieves close to the maximum possible performance, but Sidney does not. Making a minor modification to Sidney allowed it to perform as well as the explicit version.

This chapter expands on the results of Chapter 6 and presents a much more detailed picture of Sidney's behavior. It shows that the differences between the uniprocessor and multiprocessor results are a function of the rotational speed of the disks used, not the number of processors. The results show that in large part, Sidney and the explicit version achieve good performance because all of their computation is overlapped with the rotation of the disk. In the case that Sidney cannot complete its computation in one disk rotation, performance degrades. This chapter presents strong evidence for why this degradation occurs and also shows how it was overcome. The chapter also addresses the basic question of how time is spent while running **trans** and how Sidney could be sped up.

The chapter begins with a description of TPC-B, the variant of it based on Sidney, and the variant of it based on explicit persistence. This section also discusses how the synchronous write performance is measured. The next section begins with a detailed summary of the performance results. It then reviews the previously presented results, including additional information about the distributions of the values measured. It continues with a more detailed examination of the factors influencing the throughput of Sidney, including the description of how Sidney was changed to achieve the same performance as the explicit version on the faster disks. This is followed by more details about the `commit` overheads, both for Sidney and the explicit persistence version. Next, there are some results concerning the latencies of `commit`, followed by a discussion of how performance would change if disks or CPUs became faster. The chapter ends with a brief summary.

7.1 Description of Trans

Trans is based on TPC-B, an industry standard transaction processing benchmark . TPC-B first performs a very small number of bank teller operations involving transfers among various bank accounts and then commits. Designed by the Transaction Processing Council [55], TPC-B and the closely related benchmarks, TPC-A and TPC-C, are popular benchmarks for studying commercial transaction systems. These benchmarks were designed to model the performance of real transaction-based applications on small transactions. The official use of the term TPC-B for describing a particular benchmark requires an extensive certification by the Transaction Processing Council; for this reason, I call my benchmark **trans**.

Trans-initial is a slightly non-standard implementation of TPC-B in SML. It is based on the description of TPC-B in Gray [55]. It differs from the standard only in that it does not satisfy the scaling rules for database size; the database is sized to fit in physical memory. Trans-initial does a small amount of computation and then commits its changes to the disk. Trans-improved differs from trans-initial only in that it uses a slightly improved version of Sidney. The improvements are discussed below, but note that these changes to Sidney only apply to trans-improved; all other results in the dissertation are based on the initial version of Sidney.

Trans-explicit is also a variant of TPC-B. Trans-explicit was coded by Puneet Kumar in C++ and uses RVM directly to provide transactions and persistence. Unlike the versions based on Sidney, trans-explicit does no dynamic storage allocation. This difference is a further deviation from the TPC-B standard. Since it is based on RVM, so that it shares much of the same low-level mechanism as Sidney, but uses explicit techniques, trans-explicit is ideal for comparing Sidney to the explicit approach.

Trans-raw is a very simple program that writes a fixed amount of data to the disk synchronously. Since the synchronous disk write is the most expensive part of `commit`, trans-raw indicates the best performance that the TPC-B based versions might possibly achieve.

Since trans-initial and trans-improved do no collection except as a side-effect of `commit`, the GC parameters are irrelevant. For testing purposes, the benchmark is run for 10,000 transactions, resulting in total times to run the benchmark on the uniprocessor of about

3 minutes. Each `commit` transfers 56 bytes from the transitory heap into the persistent heap. As little as 56 bytes could conceivably be transferred to the disk as well, but in practice a sector, 512 bytes, is transferred by RVM. Trans-raw also writes sectors, which was found experimentally to maximize its performance. Trans-explicit is configured to match the configuration used by trans-initial; for example, both use a database of the same size.

It is worth noting that the systems studied here are much different from the commercial systems normally measured with TPC-B. In those systems, the goal generally is to maximize the total throughput of the system. This goal is achieved by using many disk arms and many concurrent transactions and using techniques such as group commit. The result is that these systems are CPU limited, not disk limited as **trans** is. Furthermore, these systems are tested on databases that do not fit in physical memory. My goal was not to compare Sidney to these commercial systems, but rather to an explicit persistence system with similar design goals and low-level implementation.

7.2 Results of Trans

For **trans** the primary difference between the uniprocessor and multiprocessor is that the multiprocessor has a disk that rotates faster, not that the multiprocessor has more than one CPU. On the uniprocessor, there are no important differences between trans-initial and trans-improved. Thus, except for the highest level results, I present only the multiprocessor data. In summary, the results of **trans** are:

Trans Throughput

On the uniprocessor, the maximum possible throughput is 60 TPS. All four variants of **trans** achieve close to this rate. On the multiprocessor, the maximum possible throughput is 90 TPS. All variants, except for trans-initial, achieve this rate; trans-initial has a throughput less than it achieved on the uniprocessor.

Synchronous Write

The time to perform the synchronous write needed for `commit` reveals that trans-explicit and trans-initial are able to overlap all computation with the disk's rotational latency. This result is also true of trans-initial on the uniprocessor, but not on the multiprocessor. That trans-initial sometimes misses disk rotations accounts for its degraded performance. This result shows that the important difference between the two machines is the rotational speed of the disks, not the differing number of processors.

From Trans-Initial to Trans-Improved

Investigating the overheads present in trans-initial suggested that eliminating the instruction cache flush on `commit`, which is not required in this benchmark, might allow the benchmark to overlap all its computation with the disk rotation. This change was successful and resulted in trans-improved.

Trans-Improved `Commit` Overheads

The dominant overhead in **trans** is the synchronous write. Further investigation of trans-improved shows that the other important overheads are all RVM-related, suggesting that either reducing the use of RVM or speeding up RVM itself is likely to be the most effective way of achieving further performance improvements. The cost of orthogonal persistence is negligible.

Trans-Explicit `Commit` Overheads

After the synchronous write, RVM is also the main overhead in trans-explicit. Again, further performance improvements are most likely to come from either improving RVM or reducing its use.

`Commit` Pauses

Investigating the pauses caused by `commit` gives conclusive evidence that trans-initial is missing revolutions on the multiprocessor.

7.2.1 Trans Throughput

This subsection examines the factors influencing the throughput of **trans**. It begins by reviewing the results of Chapter 6. The next few parts discuss why trans-initial is slower than trans-improved and explains how I changed Sidney to create trans-improved. Finally, I discuss the major overheads found in `commit`, for both trans-improved and trans-explicit.

Total Time

Chapter 6 presents the basic **trans** results in the standard way, as number of transactions per second. Here, they are presented in terms of the time per transaction, which is more useful when comparing them to the times for individual components of the overhead. Unlike the previous presentation, I also include information about the quartiles of the distribution in addition to the median.

Table 7.1 shows the result of measuring the elapsed time for **trans** on the DECstation 5000/200 uniprocessor. The format used is the standard one described in Chapter 6, except that instead of saying what collector is being measured, the particular sub-benchmark is specified. Since **trans** does no GC (except synchronously as part of `commit`), the collector used is irrelevant. Also note that all measurements shown in the chapter are of elapsed time. The CPU times are either essentially the same as the elapsed times, for example, when measuring the components of Sidney, or negligible, for example, when measuring the costs of I/O.

The disk used on the DECstation 5000/200 rotates at 3600 RPM, or 16.67 msec per revolution. Because of the layout of RVM's log on disk, at most one log write can be done per revolution, thus bounding the shortest transaction possible to be 16.67 msec or 60 TPS. The table shows that the actual programs perform at close to 17 msec, resulting in transaction rates above 50 TPS. Trans-raw has the best performance, but all of the TPC-B

Quantity Measured	Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
Total Time	Initial	17.6	100.0	-0.2	0.4
Total Time	Improved	17.7	100.0	-0.4	0.4
Total Time	Explicit	17.8	100.0	0.0	0.0
Total Time	Raw	17.0	100.0	0.0	0.0

Table 7.1: Total Times (uniprocessor)

variants achieve within 5% of trans-raw. That trans-raw is not able to always write to the disk in 16.67 msec suggests that modeling the disk performance purely in terms of rotational delay is somewhat naive. At the least an occasional seek must be done. Both the 25th and 75th percentiles are within 0.5% of the median (and for trans-explicit and trans-raw, within 0.05%), which means the distributions are tight and implies that any error in the median is quite low. Unless the quartile values are notable, I will omit discussing them.

Table 7.2 shows (in the same format as Table 7.1) the result of measuring the total time for **trans** on the multiprocessor.

The disk used on the multiprocessor rotates at 5400 RPM, or 11.11 msec per revolution, limiting the throughput to 90 TPS. All of the benchmarks except for trans-initial come close to 11 msec and thus 90 TPS. Trans-initial does not perform nearly as well; in fact it does worse than with the slower disk on the uniprocessor, achieving only 54 TPS. The next few parts attempt to explain why.

Synchronous Write

Since it differs so drastically from the other results, it is important to understand why trans-initial performs as it does. This understanding motivated the changes to Sidney that resulted in trans-improved, and may suggest other improvements. This part presents the results of measuring just the synchronous log write. This measurement is especially telling because the overheads it measures are entirely outside of Sidney and RVM and depend only on the system overheads and cost of writing the disk.

Quantity Measured	Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
Total Time	Initial	18.6	100.0	-0.9	1.7
Total Time	Improved	11.6	100.0	-0.2	0.9
Total Time	Explicit	11.6	100.0	0.0	0.0
Total Time	Raw	11.3	100.0	0.0	0.1

Table 7.2: Total Times (multiprocessor)

Quantity Measured	Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
Synchronous Write	Initial	11.3	60.0	-0.5	1.6
Synchronous Write	Improved	5.9	51.0	-0.6	1.0
Synchronous Write	Explicit	9.5	82.0	-0.8	0.6

Table 7.3: Synchronous Write (multiprocessor)

Table 7.3 shows the result of measuring the cost of the synchronous log write on the multiprocessor. The times shown correspond to how long it takes the I/O system to write the data to disk and includes both the system processing time and the time for positioning the disk head. Since trans-raw consists of only a synchronous write, its synchronous write time has already been reported as its total time.

The crucial point to note is that while both trans-improved and trans-explicit have times that are substantially less than the rotational delay of the disk, trans-initial has a time that is slightly greater than the rotational delay. Both trans-improved and trans-explicit are able to overlap their computation with the disk rotation, but at least some of the time trans-initial incurs an additional delay because of missing a revolution. If trans-initial always missed a revolution, we would expect it to take 22 msec of total time, thus it seems probable that it only misses the revolution some of the time. We will see later that there is about 4 msec of system overhead in the synchronous write, thus trans-initial having a synchronous write time very close to the rotational delay means it must miss at least some rotations.

From Trans-Initial to Trans-Improved

How was trans-initial modified to create trans-improved? The observation was that trans-initial sometimes missed a disk rotation because its computation took too long; thus the basic idea was to find some part of Sidney to speedup. Examining the cost of various parts of the implementation suggested that eliminating the instruction cache flush might have the desired effect. Recall that after a `commit`, Sidney flushes the instruction cache on the chance that code may have been copied from the transitory heap into the persistent heap. Trans-initial does not copy any code, thus I was able to safely eliminate the cache flush, creating trans-improved. As we have seen, trans-improved achieves basically the same performance as trans-explicit and trans-raw. As discussed in Chapter 5, the need to flush the cache at all is very specific to certain design decisions in SML/NJ and thus omitting it to get trans-improved does not compromise the generality of these results.

Table 7.4 shows the result of measuring the cost of flushing the instruction cache on the multiprocessor.

In trans-initial, the flush accounts for 1.5 msec per `commit` or 8% of the total execution time. Eliminating this overhead is enough to avoid missing rotations. The cost shown for trans-improved is the cost of calling the measurement code. Note that the small value leads to a much larger percentage spread in the distribution.

Quantity Measured	Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
Cache Flush	Initial	1.5	8.0	-0.3	0.4
Cache Flush	Improved	0.1	1.0	-8.9	4.8

Table 7.4: Cache Flush (multiprocessor)

Trans-Improved `Commit` Overheads

It is instructive to look more closely at the contributors to the cost of `commit`. We have already seen that the most important is the synchronous log write, accounting for about 50% of the time in trans-improved. To find out what parts of the system were responsible for the rest of the overhead, I timed the various components of trans-initial and trans-improved. Since the times for trans-initial and trans-improved were very similar, except of course for the synchronous write and the cache flush, I have omitted the numbers for trans-initial. I have also omitted components that did not contribute significantly to the overhead.

Table 7.5 shows the result of measuring the cost of a variety of overheads for trans-improved on the multiprocessor. The quantity labeled *Commit* measures the time spent in the runtime system during `commit`. Since trans-improved (and trans-initial) does no collection except as part of `commit`, the cost of `commit` is all of the runtime cost. The quantity labeled *Free Pointer* measures the time spent updating various bookkeeping aspects of Sidney just before the RVM commit, for example, to update the persistent heap's free pointer, and to record that modification with RVM. These calculations are trivial except for the calls to RVM. The quantity labeled *RVM Commit* is the time spent in RVM during the RVM commit operation before the synchronous write. The quantity labeled *Log Writes* is the time spent informing RVM of the locations found on the write log. The quantity labeled *Minor Collection* is the time spent doing minor collections in preparation for the `commit`. Finally, the quantity labeled *Commit GC* is the cost of the commit GC. Recall that the commit GC is the collection that transfers data from the transient heap to the persistent heap.

Quantity Measured	Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
Commit	Improved	11.1	96.0	-0.2	0.3
Free Pointer	Improved	2.5	21.0	-1.4	0.9
RVM Commit	Improved	1.2	11.0	-3.8	3.8
Log Writes	Improved	0.6	5.3	-1.1	3.2
Minor Collection	Improved	0.4	3.8	-1.4	4.3
Commit GC	Improved	0.3	2.9	-1.7	2.4

Table 7.5: Trans-Improved Overheads (multiprocessor)

Quantity Measured	Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
RVM Commit	Explicit	2.0	17.0	-1.9	3.6

Table 7.6: Trans-Explicit Overheads (multiprocessor)

The time spent in the runtime during `commit` is 11.1 msec, which is 0.5 msec less than the total time. This difference is the time spent in SML code and transferring between SML and the runtime. The last five entries in this table are the major subcomponents of the `commit` overhead except for the synchronous write. Summing these five quantities gives 5.0 msec, matching the 5.2 msec difference between the overall time for `commit` (11.1 msec) and the time for the synchronous write (5.9 msec). The entries labeled *Free Pointer*, *RVM Commit*, and *Log Writes* measure parts of the system that primarily involve RVM and account for 4.3 msec of the overhead. Thus, to improve Sidney's performance even more, the best alternatives are either to reduce Sidney's use of RVM or to speed up RVM itself. Of the other costs, the need to do a minor collection is purely a matter of implementation convenience. The only cost that comes directly from using orthogonal persistence, Commit GC, moving transient data into the persistent heap, is almost negligible, 0.3 msec.

Trans-Explicit `Commit` Overheads

It is also useful to look at the cost of trans-explicit, excluding the synchronous write. This closer look allows us to make a more detailed comparison of Sidney and the explicit approach.

Table 7.6 shows the result of measuring the cost of trans-explicit excluding the synchronous write on the multiprocessor.

From the difference between the total time (and synchronous write) time for trans-raw, 11.3 msec, and the synchronous write time for trans-explicit, 9.5 msec, we can estimate that the CPU overhead that overlaps with the disk rotation is 1.8 msec, which matches the 2.0 msec for this measurement reasonably closely. Almost all of the computation in this case is due to RVM; so like Sidney, to improve the explicit approach's performance, the best approach is either to reduce the use of RVM or to speed up RVM itself.

7.2.2 Trans Latencies

The overview in Chapter 6 did not address the latencies present in the **trans** benchmark, in part because the main reason for discussing latencies there was to understand the GC pauses. However, studying them does give further evidence about why trans-initial suffers a performance degradation when the disk speed is increased. They also give some clues about system overheads. For these reasons I examine them here.

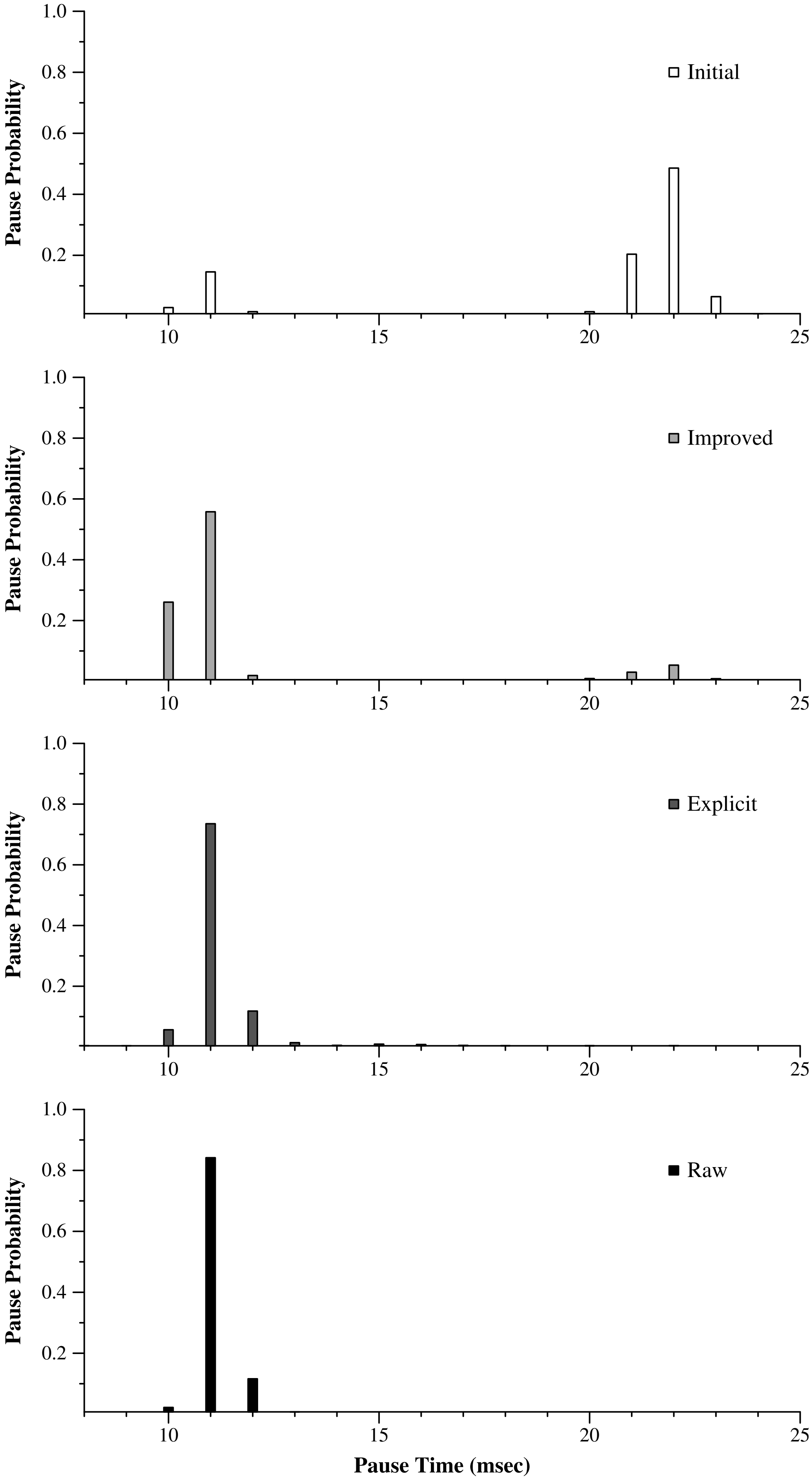


Figure 7.7: Transaction Commit Pause Times (multiprocessor)

`Commit` Pauses

The first measurement of interest is the latency of the `commit` operation. If the `commit` times depend strongly on the rotational speed, then these times should be closely clustered around multiples of the rotational frequency. If trans-initial is missing some but not all rotations it should show a bimodal distribution of pause times. These points assume the transactions all take the same amount of time, which is true for the benchmark.

Figure 7.7 shows the result of measuring the elapsed time taken by individual commits for all of **trans** on the multiprocessor. The x-axis is the length of the pause in milliseconds. The pauses have been placed into bins 1 msec wide. The y-axis is the probability of having a pause of the given duration. To facilitate comparison, each plot has the same axes.

The pauses for trans-initial have a bimodal distribution, with one peak at 11 msec and the other at 22 msec. This is strong evidence that trans-initial does miss some but not all disk rotations. If the peaks near 22 msec occurred with about twice the probability of those at 11 msec, then the average time for a transaction would be 18 msec as was measured. I believe that the fact that the pauses at 22 msec are closer to four times as frequent is a result of the measurements perturbing the results slightly. There is more evidence for this later. The other benchmarks show only a peak at 11 msec, except for trans-improved which shows a small but noticeable peak at 22 msec. I believe this peak is also a result of perturbations cause by the measurements, since the earlier measurements show clearly that the time for a transaction in trans-improved is the same as for trans-explicit. These peaks' presence suggests that the improvements to trans-improved are just enough to allow the benchmark to achieve maximum performance.

Synchronous Write Pauses

It is instructive to look at the pause times for the synchronous write. This time does not include any of the overheads due to Sidney or RVM, only system overheads and disk-related delays. Examining them gives more evidence concerning the system-related overheads.

Figure 7.8 shows the results of measuring the time taken by the synchronous log writes for trans-initial and trans-improved; the other benchmarks do not contain any additional information and have been omitted. The axes are the same as for Figure 7.7 except that the x-axis has a different origin.

Surprisingly, trans-initial shows only a single peak at 15 msec, when we would expect it to be bimodal. For this measurement, trans-initial achieves about 45 TPS, which means the `commit` time is 22 msec or two rotations. This is further evidence that timing this part of the code is perturbing it slightly, but enough to change these results, which are very sensitive to small changes in the timings. The peak at 15 msec suggests that there are 4 msec of system overhead followed by 11 msec of rotational delay. I present further evidence of this in the next section. Trans-improved has pauses of 5 to 6 msec and a much smaller number around 15 msec, supporting the claim that it writes a log record on almost every disk rotation. If we assume (as before) that for most of the measurements, two-thirds of the time trans-initial takes 15 msec and one-third of the time it takes 5 msec then the average time for the synchronous write is 11.5 msec, matching the measured average, 11.3 msec.

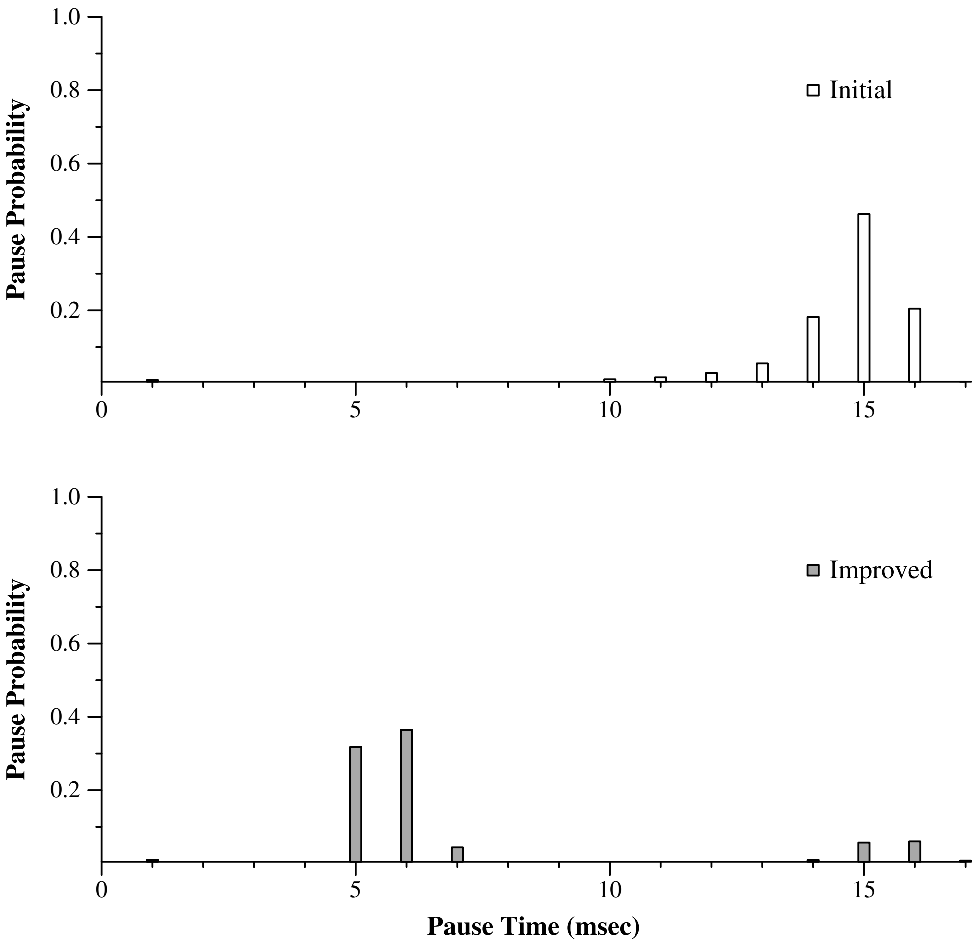


Figure 7.8: Synchronous Log Write Pause Times (multiprocessor)

7.2.3 Discussion

The overheads of trans-improved and trans-explicit are quite different and yet they each achieve the same overall performance. They both achieve good performance because both manage to overlap essentially all computation with the disk rotational delay. Trans-initial is very close to the borderline between one and two rotations and thus its computational overhead is close to the maximum that can be overlapped. The computation overhead can be divided into two parts: the user computation that takes place in Sidney and RVM, and the system overhead that takes place during the synchronous write to the log. The user overhead can be found by subtracting the time spent in the synchronous write (which includes the system overhead and the time waiting for the disk to rotate) from the overall cost per transaction. These overheads are as follows: trans-initial, 7.3 msec; trans-improved, 5.7 msec; trans-explicit, 2.1 msec. The difference between trans-initial and trans-improved is 1.6 msec, corresponding to the cost of flushing the cache. Assuming that 7 msec is the maximum user overhead that avoids the extra rotation, the system overhead is about 4 msec. The cost of the synchronous write for trans-initial also reflects this system overhead as discussed above.

Estimating the system overhead allows us to speculate about what would happen if some part of the system got faster. Since the system is limited by the disk rotational speed, increasing the processor performance would not help system performance, unless the disks also got faster. However, since Sidney is either missing or close to missing revolutions, speeding up the disk may not help either. Speeding up the disk from 3600 RPM to 5400 RPM resulted in a slight slowdown for trans-initial. Trans-explicit could benefit from a faster disk. Its user overhead is 2 msec; assuming the system overheads are the same as for Sidney (a reasonable assumption, since both write one sector), 4 msec, the total overhead is 6 msec. Thus the explicit version could take advantage of a disk with a rotational latency of 6 msec or 10000 RPM, allowing it to achieve a rate of 160 TPS. The fastest disks currently available are 7200 RPM and this speed is not increasing rapidly, so in fact the current limit is probably 120 TPS. Modest speedups, achievable simply by moving to a more modern CPU should allow Sidney to achieve the maximum performance on these disks as well.

One thing that is troubling is how sensitive the results for Sidney are to small changes in the `commit` overheads. This sensitivity is basically an artifact of the artificial nature of the benchmark. It is caused by the fact that all of the transactions do exactly the same amount of work and so are highly correlated with the disk rotations. The correlation causes the overall performance to be a step function. If the commits were uncorrelated with the rotation then roughly speaking, the average time for a transaction would be the average time taken by the computation plus the average rotational delay. Thus the performance would change smoothly with increasing computational overhead. Using the overheads computed above and an average rotational delay of 5.5 msec, this would result in the following performance: trans-raw, 9.5 msec per transaction or 105 TPS; trans-explicit, 11.5 msec per transaction or 87 TPS; trans-improved, 15.2 msec per transaction or 66 TPS; and trans-initial, 16.8 msec per transaction or 59 TPS. Even if the beginnings of the transactions are highly correlated with the disk rotation, as they might be in a system with very small transactions, any variance in the length of the transaction will create a distribution of possible transaction times. Exactly how many of these times fall in any particular multiple of the disk rotation time will determine how many miss greater or fewer rotations. This effect is seen in the current test where some of the trans-initial transactions do not miss revolutions. The tight distributions of times caused by doing exactly the same work over and over makes the transition be close to a step function.

One other issue arises. Instead of waiting for the disk to rotate to the end of the log, it would be possible to write the log to the next block that was free and could be written, allowing the rotational latency to be avoided. This is called *sector skewing*. This change would complicate the design of RVM's log manager since it would have to keep track of what parts of the disk were used, as well as predict where the next unused block would be on the disk. This kind of optimization might be worthwhile in a commercial system where the rotational latency is a key issue, but it seems doubtful that it is a good idea for RVM. The effectiveness of such an optimization is limited by the system overhead and RVM overhead. At most it would allow trans-explicit to achieve 7.5 msec per transaction or 133 TPS. Since trans-improved already writes a log record almost as frequently as possible, little speedup would be seen there. For longer transactions, it would have an insignificant impact.

7.3 Summary

This chapter presented the details of the **trans** benchmark. A variant of the industry standard benchmark TPC-B, **trans** measures the throughput of small transactions. The measurements presented here compare Sidney to a system that uses an explicit model of persistence and to the maximum rate achievable just by writing the log as fast as possible. In all but one case, Sidney performs competitively with all the other benchmarks. In general, the benchmarks are able to overlap all their computation with the rotational delay of the disk, which allows them to perform a `commit` on each rotation. In the one exceptional case, where Sidney sometimes misses a disk rotation, it shows greatly degraded performance. This degradation was eliminated by making a small change to Sidney.

Chapter 8

The Coda Benchmark

A central claim of my thesis is that GC can have performance equivalent to explicit techniques for persistent storage management. The goal of this chapter is to show that this claim is true in terms of throughput; other benchmarks address the question of collection latencies. It would be impossible to show that the performance of GC and explicit allocation is equivalent in general; instead, I will show that for a real application, the two are comparable.

Comparing GC and malloc-and-free is hard because it is generally not possible to run the same program using both techniques. A further difficulty is that very few applications have been written using general-purpose persistence of any kind, making it hard to know what a typical application's storage management demands are. I solved this problem by using a trace-driven approach in which traces of a large application provide realistic input to a program designed to allow a direct comparison of GC and malloc-and-free. I traced the persistent memory management behavior of the Coda file system, a large systems application. The traces are then replayed using either Sidney or Coda's Recoverable Data Segment (RDS) malloc-and-free and performing the same memory management operations as Coda. The design of the trace replay program allows it to avoid the problems that generally preclude applications from using both techniques. I also replay the trace using stop-and-copy collection, so that I can compare the costs of Sidney and RDS to other implicit approaches.

The results of running **coda** support the thesis. Sidney is consistently faster than the persistent malloc-and-free. This chapter shows that Sidney performs better because it has lower persistence-related overheads, both in terms of the CPU cycles used by RVM and the cost of I/O. Sidney also outperforms stop-and-copy collection. In this case, Sidney's main advantage is that it is able to overlap computation with I/O, whereas the stop-and-copy collector must be synchronous. The performance gain due to I/O overlap is significant enough that concurrent collection outperforms stop-and-copy collection, not only on the multiprocessor, but also on the uniprocessor.

One result should not be overlooked. Generating the trace took approximately one month of fairly heavy Coda use. Replaying the trace takes at most a few minutes, no matter what method is used. Thus for Coda, the impact of persistent memory management on throughput is minimal, and small performance differences between GC and malloc-and-free are irrelevant. This argues strongly for the use of GC, since it would provide greater safety and programmer convenience. Of course, this argument is only true if collection pause times can be effectively dealt with. This issue is addressed in the following two chapters.

The first section of this chapter describes the benchmark itself. This description begins with the key systems used in the study: Coda, RDS, and Sidney. I then describe trace generation and replay. The second section contains the results of running **coda**. The results begin by examining how Coda uses the persistent heap. I then present the throughput results for malloc-and-free, followed by those for Sidney, and conclude with those for stop-and-copy collection. Due to the nature of the trace replay program, no latency measurements are presented. The chapter concludes with a brief summary.

8.1 Description of Coda

In designing the **coda** benchmark, I had a number of goals. First, I wanted to make as direct a comparison of Sidney to malloc-and-free as possible. Without a direct comparison, it would be difficult to support the claims of the thesis. Second, I wanted the benchmark to reflect the memory management behavior of an existing, realistic application that used persistence. Since few persistent applications exist, I could not count on past experience to understand what performance issues were important. By basing the study on a real application, at least I captured the performance issues that are important to that application. Finally, to the extent that the benchmark needed to be biased at all, I wanted to bias it in favor of malloc-and-free.

Simply running the same programs using persistent GC and malloc-and-free would be the most direct way to compare the performance of the two techniques. Unfortunately, applications are designed to use either explicit or implicit storage management. The choice of technique is generally dictated by the programming language or language implementation being used, so switching between the techniques would require recoding the application. Some direct studies have been done using conservative garbage collectors [63]; in general, they show that GC can have performance that is competitive with a variety of malloc-and-free implementations, usually at the cost of using more memory. The copying techniques used by Sidney are not compatible with conservative collection, so it was impossible to duplicate these studies using Sidney.

Since it was not possible to compare the two techniques in the most direct way, I used the trace-based approach discussed in this section. I first describe the key systems used for trace generation and replay: Coda, a reliable distributed file system, which was used for trace generation; RDS, a persistent malloc-and-free for RVM, which is used by Coda and also as the malloc-and-free for trace replay; and Sidney, which has a few differences from the version used in the rest of the thesis. Finally, I discuss how tracing was added to Coda and how the trace replay program works.

8.1.1 The Coda File System

I chose to trace the Coda file system [50] because it is the largest persistent application in production use to which I had access, and because its persistence mechanism is also based on RVM, facilitating the comparison to Sidney.

Coda is a reliable distributed file system that makes extensive use of general-purpose persistence in its implementation. It is a large system in daily use by a significant user community, making it a good candidate for studying the performance impact of persistent storage management alternatives. Coda achieves high reliability through the use of server replication and support for disconnected operation. The implementation of both of these key features makes significant use of persistent storage. Persistent storage is also used to store file system meta-data such as directory information. The implementors of Coda consider persistence and transactions to be essential to Coda's implementation [31].

RVM was designed to provide efficient transactional update of persistent data for Coda; its basic no-frills design also made it a good base upon which to build Sidney. Coda makes use of some statically-allocated RVM storage, but most RVM-backed storage is dynamically allocated using RDS malloc-and-free. The designers of Coda attempted to be conservative in their use of persistent storage. The results of this study suggest that they were successful.

8.1.2 RDS

RDS adds services to RVM that are required to implement malloc-and-free for a persistent heap and was implemented by David Steere. Together, RVM and RDS comprise an explicit persistent storage management system. The programmer must decide what data needs to be persistent, allocate and deallocate it manually, and explicitly log changes to the data to RVM. The facilities added by RDS include the ability to initialize a heap and to load an existing heap into memory at a fixed location, as well as versions of `malloc` and `free` that allocate storage backed by RVM. Since RDS provides an explicit model of persistence based on RVM, it makes a good contrast to Sidney.

Many aspects of the RDS malloc-and-free implementation are the same as for a conventional malloc-and-free. RDS allocates storage from a series of free lists holding blocks of storage that are a multiple of a fixed block-size. The number of free lists and the block-size are fixed at heap initialization. To allocate storage, the allocator rounds the user request up to a multiple of the block-size and uses the number of blocks requested as an index into an array of free lists. The last free list in the array contains a list of blocks that are larger than the block-size times the number of free lists. There are three words of overhead added to each user request. One word at the beginning of the block is used for a header that records whether the block is allocated or not. The next word is used to record the size of the block, and the last word of the block is used as a guard. Both the header and guard words are checked for validity when RDS manipulates the storage; this check aids in detecting any memory overwrites that may have occurred. When a user makes a request, if a block of the appropriate size is not available a larger block is split into two blocks, one sized to fit the user request. If the heap becomes so fragmented that a request cannot be satisfied, the system attempts to coalesce the free blocks into a block that is large enough to satisfy the request. Coalescing can lead to unbounded pauses, although such attempts are rare. If coalescing fails to create a large enough free block, the `malloc` may fail even though there is enough free (but fragmented) memory in the heap. The implementors of Coda have observed both very long pauses due to coalescing and allocation failures due to fragmentation.

Some aspects of the RDS implementation are unique to transactions and persistence. The most obvious is that any changes to the RDS data structures must be recorded with RVM so that they become permanent at the time of `commit`. An important consequence of logging changes to RVM is that the overhead for persistent malloc-and-free is much higher than for a conventional implementation. I examine the effects of this overhead in some detail in this chapter. The RDS heap contains pointers, so when it is loaded into memory it must always be placed in the same place in memory. The same issue arises in Sidney, which also attempts to load its heap into its old location; if that is not possible, it can load it into a new place and then correct the pointers. A problem with RDS concerns the interaction of `free` and transaction abort. Due to certain details of the RVM and RDS implementations and their interactions with threads, if a transaction does a `free` and then aborts, it is possible for the `free` not to be successfully undone. This results in storage being freed that is still in use, a serious error. RDS uses the following solution: instead of actually freeing storage, a version of `free` is used that records the intention to free the storage in a list. Then when the transaction commits, this intentions list is used to actually perform the frees. If the transaction aborts, the list is simply discarded. A similar problem arises in `malloc`; unfortunately it is not possible to delay the `malloc`. However having a `malloc` performed and then not successfully aborted only creates a storage leak, rather than freeing storage that is in use; RDS simply tolerates this problem. The need to record the frees and then replay them also adds overhead to the implementation.

8.1.3 Sidney

For **coda**, some aspects of Sidney are different from those described in the first part of the dissertation. Normally all allocation is done by the SML program in the transitory heap. Since trace replay is done entirely in the runtime, no SML code executes and there is no transitory heap. Instead, allocation is done directly in the persistent heap by a simple allocation routine that just changes the persistent heap free pointer. This difference means that the **coda** benchmark measurements do not contain the overhead of orthogonal persistence. Not including orthogonal persistence keeps the issues of collection separate from those of orthogonal persistence. Direct allocation is a fair comparison because RDS also allocates directly in the persistent heap. This version of Sidney's implementation overemphasizes the cost of allocation, because normally allocation is done in-line by a few instructions, rather than by a function call. Unlike RDS, allocation does not need to log any changes to RVM. Instead all newly allocated data is made persistent upon `commit`.

8.1.4 Trace Generation

I added two facilities to RDS to support tracing. The first is the ability to record the current state of the RDS heap in a log file. The heap state is dumped in a form that allows the exact location and sizes of all objects to be recreated as well as the exact state of various RDS internal data structures, such as its free lists, to be recreated. The contents of the allocated storage are not recorded. The second facility is the ability to record the size and location of each `malloc` and `free`. For `malloc`, both the amount of storage requested and the amount returned are recorded. For `free`, when a `commit` occurs, if an intentions list exists its entries are added to the trace, and otherwise nothing is added. Together these facilities allow me to exactly reconstruct an RDS heap and to reproduce exactly the memory management operations that Coda originally performed.

I added the modified RDS to both the Coda client and server, along with code to control dumping the heap and to turn tracing on and off. The trace used in this study was generated from the Coda client program and represents the allocation of approximately 1.1 MB of persistent storage. The trace was collected over the period of approximately one month and represents my own extensive use of Coda for program development and thesis writing. It is worth noting here that these activities are quite typical of most uses of Coda, so it seems likely that the trace generated represents fairly typical Coda behavior.

8.1.5 Trace Replay

The SML/NJ runtime is a C program that includes the implementation of Sidney. I modified the runtime so that in addition to acting as the runtime for SML, it can also read the heap dumps, recreate the RDS heap, and replay the traces using either Sidney or RDS.

When measuring RDS, the runtime reads the heap description and creates a new heap that is an exact duplicate of the original heap's layout. The test program then reads the trace and performs exactly the same sequence of memory management operations as Coda did when the trace was generated. One important way trace replay differs from the original execution is in the frequency of commits. The trace itself only records commits during transactions in which data was freed. Since commits are very expensive, even if only the commits that can be identified in the trace are performed, they dominate the performance to an extent that makes it difficult to measure the storage management overhead. Instead, a `commit` is done only after a constant number of commits recorded in the trace have been replayed. This constant was chosen by finding the number of commits that minimized the cost of replaying the trace with RDS malloc-and-free.

When measuring collection, how to replay the trace is more difficult. The problem is that in a garbage-collected system there is no `free` operation. Thus when a `free` is replayed it is unclear exactly what to do. Basically what is needed is to make the data unreachable so that when the next collection occurs the data will be collected. Achieving this result is simple. When a data item is allocated, it is placed in a hash table that is reachable from the persistent root. When a `free` occurs, the corresponding item is removed from the hash table making it unreachable. When the garbage collector eventually runs, the storage is reclaimed. To make the malloc-and-free code as similar as possible, the same basic trace replay and hash table code are used in both cases. The need to do anything when freeing storage adds overhead to GC that would normally not be present.

When replaying the trace, the stop-and-copy collector is run when some allocation causes the free pointer to exceed the GC threshold. The concurrent collector is triggered when an allocation exceeds the same limit, although the flip occurs at a later time. Due to the artificial nature of the test, if the collection were allowed to be as asynchronous as possible, the collector would not have enough cycles to achieve even one flip while replaying the trace. Thus, I have included most of the collector copying work in-line with the trace replay while still allowing all I/O to be asynchronous. The in-line copying slows down the client thread, but allows the asynchronous I/O phase of the collection to begin earlier. This overhead makes the concurrent collector costs higher than would be encountered when there is enough client calculation to allow most of the copying to be asynchronous, thus significantly biasing the measurement in favor of malloc-and-free and stop-and-copy collection. In practice, almost none of the copying is done concurrently.

8.2 Results of Coda

Showing that in general GC has as good performance as malloc-and-free for persistent storage management is not a goal of this experiment, nor would such a demonstration be possible. The goal is to compare Sidney to an explicit allocator taken from a real system. The key question is whether collection could replace malloc-and-free without an unacceptable performance impact. I also use **coda** to compare Sidney to stop-and-copy collection to see if Sidney has any performance advantages beyond shorter pause times over stop-and-copy collection. In addition to the basic high-level results, I present a number of detailed measurements that help to explain the high-level results.

I do not present results for collection pause times. The nature of the trace study makes it uninteresting to study the pauses created by GC, because they are artificially low. In particular, the lack of a transitory heap eliminates the most important contribution to the collection pause time. Other benchmarks presented in the following chapters provide evidence that the collection pause times are acceptable.

The section begins by examining how Coda uses the persistent heap as captured by the data in the trace. It provides basic information about how the livesize varies as new data is allocated. It also allows us to understand how much storage both GC and malloc-and-free waste. The throughput results follow. First are the results for RDS, including measurements of overall throughput, overhead for trace replay, `commit` cost, and the cost of RDS without RVM. Next are the results for Sidney, covering the same measurements as for RDS and including a comparison to the RDS results. The final results are for stop-and-copy collection, including the same basic measurements and a comparison to RDS and Sidney.

In summary, the results of this section are:

Heap Usage

The trace used represents the allocation of about 1.1 MB of persistent storage. The livesize of the persistent heap varies from 2.5 MB when the trace begins to 2.9 MB when it ends. The increase is roughly monotonic. The overhead for header words for GC is about 1% of the total storage, while RDS uses about 10% extra storage. The larger amount is because RDS `malloc` rounds requests to a fixed block size. I did not attempt to measure the external fragmentation for RDS or the size of the physical memory working set.

Malloc-and-Free Throughput

On the uniprocessor, the total elapsed time to replay the trace using RDS was 70 sec, of which about 30 sec is I/O. `Commit` accounts for 43 sec of the elapsed time and all of the I/O overhead. The cost of the malloc-and-frees is 23 sec. Disabling RVM reduced the cost of the malloc-and-frees to 1 sec. This large reduction shows that almost all of the cost of RDS malloc-and-free comes from supporting persistence and transactions. The multiprocessor shows similar results, except that it has greatly reduced I/O costs.

Sidney Throughput

The time to process the trace using Sidney varies depending on how many flips the collector does. In turn the number of flips is controlled by setting the GC threshold. On the uniprocessor, even at the smallest threshold and with the greatest number of flips, Sidney takes only 60 sec to process the trace, well below the time for RDS. When flips occur every 500 KB of allocation, at the two flip mark here, Sidney is 3.5 times faster than RDS. On average, each collection takes 6 sec, while `commit` takes about 6 sec and allocation about 1 sec. The cost of `commit` shows a slight dependence on the number of flips, which is due to sharing the CPU with the collector. Eliminating RVM makes the cost of `commit` zero, that of allocation 0.7 sec, and the cost per GC 2.2 sec. Thus again the majority of the overhead is due to support for persistence and transactions. The multiprocessor results are similar, except that again I/O is more efficient and, since the CPU is not shared, `commit` is independent of the number of flips.

Stop-and-Copy Throughput

The results for stop-and-copy collection are similar to those for Sidney, except higher because stop-and-copy cannot overlap computation with I/O. The break-even point between RDS occurs at a GC threshold between 100 KB and 200 KB, which is much smaller than would be used in practice. The cost of `commit` and allocation is the same as for Sidney, while the cost per collection is 7.3 sec. Without RVM, the cost per collection is 2.0 sec. The measurements are probably not sensitive enough to make a clear comparison between stop-and-copy and concurrent collection for this case.

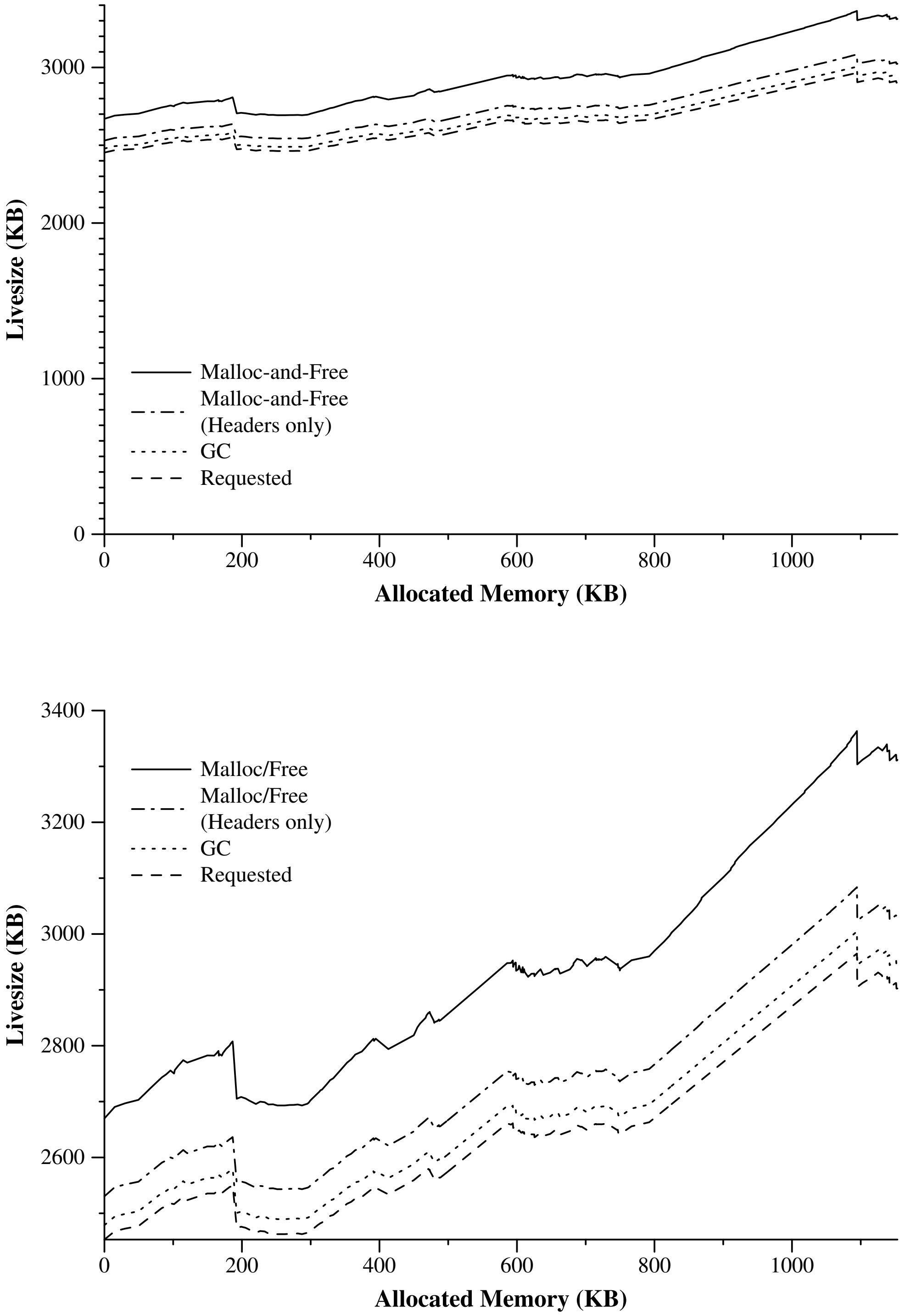


Figure 8.1: Coda Heap Usage

8.2.1 Heap Usage

Examining how the livesize changes as memory is allocated in the Coda heap gives us insight into two important issues. First, the livesize determines the cost of a collection and how it changes over time, and also tells how much deallocation a malloc-and-free implementation must do. Second, both GC and malloc-and-free have some space wasted because of bookkeeping overhead and for malloc-and-free forcing allocations to be in multiples of the block-size. The traces have enough information to calculate this wasted space and how it changes with allocation.

Figure 8.1 plots the livesize of the Coda heap versus the amount of data allocated. The x-axis specifies the amount of data allocated in kilobytes starting from the beginning of the trace. The y-axis indicates the number of kilobytes of live data in the heap. There are four quantities plotted: the top line is the amount of data actually used by RDS; the second line is the amount RDS would use if it did not round user requests up to a fixed size; the third line is the amount used by GC; and the fourth line is the amount of data actually requested by the user. The bottom plot shows the same data, but with a different y-axis to make it easier to see the differences in the results.

The maximum x-axis value shows the amount of data allocated during the trace, a little more than 1.1 MB. For GC, the average amount of data in use is 2.7 MB, the minimum is 2.5 MB, and the maximum is 2.9 MB. The different in size between the maximum and the minimum is about 15%; thus we would expect that during trace replay individual collections would vary in cost by at most 15%.

Figure 8.2 shows the amount of storage wasted for both malloc-and-free and GC. The x-axis again specifies the amount of data allocated in kilobytes. The y-axis specifies the waste or difference between the amount requested and the amount allocated in kilobytes for both GC and malloc-and-free. The top curve is malloc-and-free and the bottom one is GC.

GC wastes an average of 33 KB, or approximately 1% of the allocated storage. Since the overhead for garbage collected storage is four bytes per object, we can deduce that the objects in the heap average approximately 400 bytes, further implying that about 7,500 objects are allocated. Malloc-and-free wastes an average of 294 KB, or about 10% of the total storage. This average implies an overhead of approximately 40 bytes per object, of which 12 bytes are due to headers, and the remaining 28 bytes per object are due to rounding up to a block size.

Although these measurements show that RDS has a larger fixed space overhead than Sidney, they do not really give an idea of how much physical memory each technique needs to run effectively. In the case of GC, the actual storage used will increase until a collection is triggered. Delaying triggering collection will increase the storage needed while reducing the time spent collecting. When the collector is triggered, it will need space for to-space, which for the simple two-space collector used here will be the livesize. When the collection completes, the from-space will no longer be accessed, causing it to leave the working set. For malloc-and-free, the working set needed by the program may be much larger than the sizes seen here. This is because the allocator does not attempt to cluster allocations on pages, and thus some pages in the working set will contain a mixture of allocated and unallocated data. To quantify this effect would require a much more careful study of the

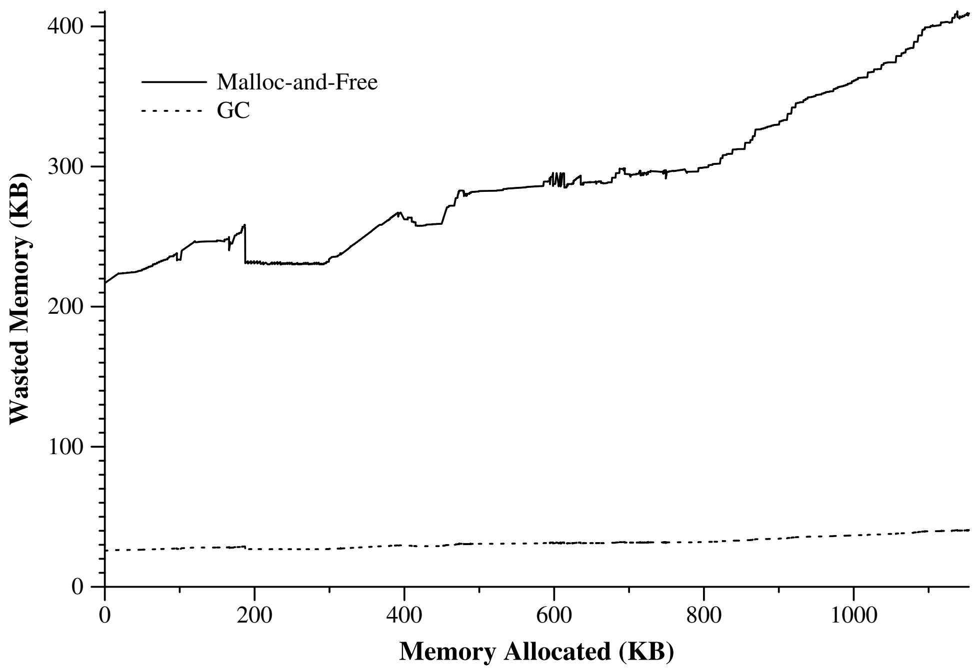


Figure 8.2: Coda Heap Waste

virtual memory aspects of RDS, but since the heap used to generate this trace contains a maximum of 8 MB, this is an upper bound on this effect as it is impossible for RDS to touch more pages than are in its heap.

The overheads found here only account for the space wasted by headers and rounding up to fixed block sizes. These are normally referred to as internal fragmentation. Another issue that is not addressed here is the extent to which the RDS heap is externally fragmented. External fragmentation refers to the fact that even if the heap has enough free space to satisfy some request, the request may fail because no contiguous block of free space exists. Since copying collection compacts the heap and always allocates from a single maximally sized piece of free storage, it does not suffer from this effect. RDS attempts to avoid this problem by coalescing free blocks when needed, but it still may suffer from this problem. How to characterize external fragmentation is a matter of debate; since Coda's heap usage is of secondary importance to me, I decided not to attempt any measurement of this type.

8.2.2 Malloc-and-Free Throughput

Since they are independent of the GC threshold, the simplest performance results to understand are those for RDS malloc-and-free. The most significant of these is total time to replay the trace. I also measured the overhead to replay the trace without actually doing any mallocs, frees, or commits, as well as the cost of `commit` alone. These measurements allow me to find the combined cost of `malloc` and `free`. Finally, I also measured the cost of replaying the trace when all RVM operations were replaced by null calls that just immediately return successfully. This measurement allows us to understand what portion of the

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
Total Time		RDS	70.1	100.0	-0.7	0.9
Total Time	CPU	RDS	40.2	100.0	-0.4	0.5
Trace Overhead		RDS	3.3	100.0	-0.2	0.6
Trace Overhead	CPU	RDS	3.3	100.0	-0.5	0.0
Commit		RDS	43.2	61.0	-0.9	0.4
Commit	CPU	RDS	13.6	33.0	-0.8	0.2
No RVM		RDS	4.2	100.0	-0.7	11.0
No RVM	CPU	RDS	4.2	100.0	-0.4	0.4

Table 8.3: RDS Times (uniprocessor)

cost of malloc-and-free is due to RVM, and thus roughly speaking the cost of persistence.

Table 8.3 shows the result of measuring the throughput of RDS on the uniprocessor. The measurement labeled *Total Time* is the total time to run the benchmark. The measurement labeled *Trace Overhead* gives the time to process the trace without doing any mallocs, frees, or commits. The measurement labeled *Commit* gives the time spent in `commit` alone. Finally, the measurement labeled *No RVM* is the time to process the trace without RVM.

RDS takes 70.1 sec elapsed (40.2 sec CPU). Since in this case RDS is single-threaded, we can conclude it spends 29.9 sec doing I/O. The overhead to replay the trace is 3.3 sec. RDS spends 43.2 sec elapsed (13.6 sec CPU) doing `commit`. The difference, 29.6 sec, accounts for essentially all the I/O. Individual calls to `malloc` and `free` are too short to measure accurately, but by subtracting the trace overhead and `commit` cost from the total we can conclude that RDS spends 23.6 sec elapsed (23.3 sec CPU) doing `malloc` and `free` combined. When all calls to RVM are eliminated, RDS takes 4.2 sec to process the trace. The trace overhead is independent of the use of RVM; without RVM the cost of `commit` is zero. Thus, we can find the cost of malloc-and-free by subtracting the trace overhead from the base cost. The result is that without RVM, RDS spends 0.9 sec doing `malloc` and `free`. With RVM the cost of malloc-and-free is more than a factor of 25 more expensive. Persistence is not cheap.

Table 8.4 shows the results of the throughput measurements for RDS for the multiprocessor in the same format as Table 8.3 .

To replay the trace, RDS takes 39.8 sec elapsed (34.3 sec CPU) and spends 5.5 sec doing I/O. The trace overhead is 4.5 sec. `Commit` accounts for 16.3 sec elapsed (10.7 sec CPU) and accounts for all of the I/O. The total cost of `malloc` and `free` is 19.0 sec. Notice that the overhead for `malloc` and `free` is similar for the two machines as is the CPU time for `commit`. However, the I/O time for `commit` differs by almost a factor of six. The multiprocessor's better I/O system has a significant effect on Sidney's performance, an effect that will also be observed in later benchmarks. Finally, without RVM, the cost of replaying the trace is 5.2 sec, of which 0.7 sec is spent actually doing allocation and deallocation. Thus, `malloc` and `free` are more than a factor of 25 cheaper without persistence.

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
Total Time		RDS	39.8	100.0	-2.1	1.7
Total Time	CPU	RDS	34.3	100.0	-2.5	1.8
Trace Overhead		RDS	4.5	100.0	-1.8	2.1
Trace Overhead	CPU	RDS	4.5	100.0	-1.9	2.1
Commit		RDS	16.3	41.0	-0.8	0.5
Commit	CPU	RDS	10.7	32.0	-0.7	0.9
No RVM		RDS	5.2	100.0	-5.8	2.4
No RVM	CPU	RDS	5.2	100.0	-5.6	2.5

Table 8.4: RDS Times (multiprocessor)

8.2.3 Sidney Throughput

The next measurements are of Sidney's throughput. Essentially the same measurements were done as for RDS, but since Sidney's performance changes with GC threshold, the threshold was also varied. Since the collector is nondeterministic, a given GC threshold can result in the collector flipping a variable number of times. The GC threshold determines the frequency with which a given number of flips occur. In analyzing and presenting the results, I have eliminated flip counts that occur with a frequency of less than 5%. The variable number of flips also makes it more difficult to present the data, since the independent variable is the GC threshold, but the throughput depends primarily on the number of flips and only indirectly on the number of GC threshold. The solution I chose is to plot throughput versus number of flips, but rather than drawing circles or boxes to show the value, using the value of the GC threshold. This makes the main trends clear, and also makes it easy to see how, for any given number of flips, the throughput depends on the GC threshold.

Total Time

Figure 8.5 shows the dependence of Sidney's throughput on the GC threshold and the number of flips achieved by the collector. The quantity appearing in the plots is the GC threshold in kilobytes. The x-axis is number of flips the collector achieved. The y-axis is the elapsed time in seconds for Sidney to replay the trace for a given threshold and number of flips. The dotted line marks the cost of RDS under the same conditions. This plot also includes error bars that plot the values of the 25th and 75th percentiles. The error bars have been omitted from subsequent plots. Since the distributions are tight, the error bars are not particularly informative and they make it harder to read the graphs.

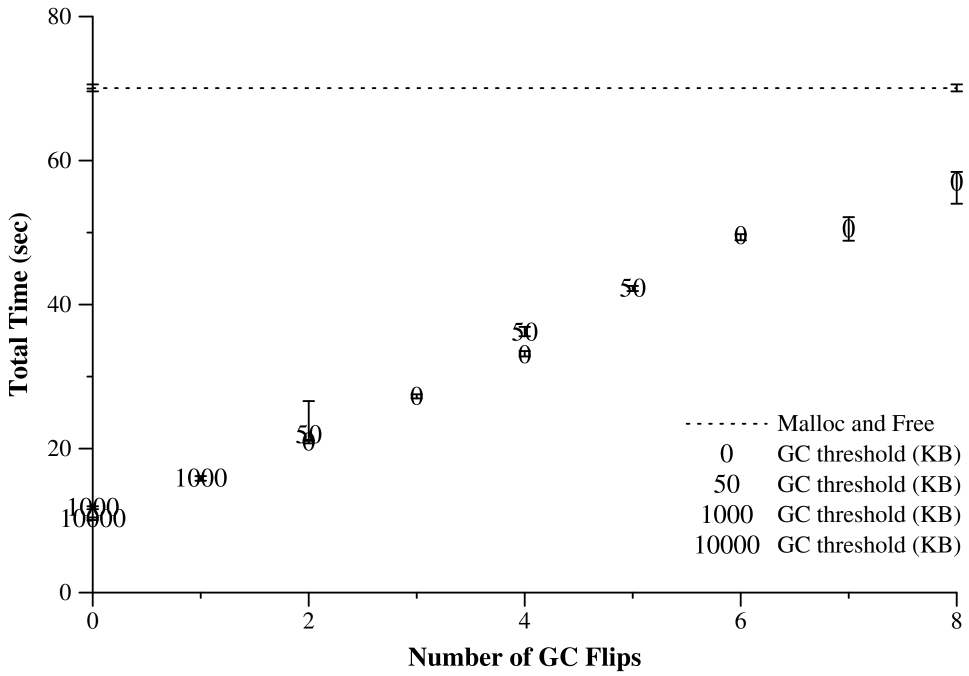


Figure 8.5: Sidney Times versus GC Flips (uniprocessor)

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
No GC		Sidney	10.20	100.0	-1.4	2.0
No GC	CPU	Sidney	5.4	100.0	-0.9	0.3
Trace Overhead		Sidney	3.1	100.0	-0.2	0.4
Trace Overhead	CPU	Sidney	3.1	100.0	0.0	0.5

Table 8.6: Sidney Times (uniprocessor)

Table 8.6 shows the results of measuring some components of Sidney's overhead in the standard form. The measurement labeled *No GC* is the total time for a GC threshold of 10,000 KB when no collection occurs. The measurement labeled *Trace Overhead* shows the cost of replaying the trace without allocating storage, doing commits, or doing collections.

Even in the most expensive case, when a GC threshold of 0 KB results in 8 flips, Sidney is more than 20% faster than RDS. When a flip occurs every 500 KB of allocation, which for real programs would be frequent and corresponds to 2 flips for this trace, Sidney is a factor of 3.5 faster than RDS. From the measurements, we see that the elapsed time is primarily a function of the number of flips; the GC threshold has an indirect effect on cost by determining the range (and frequency) of possible flips. The fact that a threshold of 0 KB results in more entries than one of 50 KB only reflects the length of the trace. The slope of the line gives the cost of collection, 5.8 sec per GC. When no collection is done,

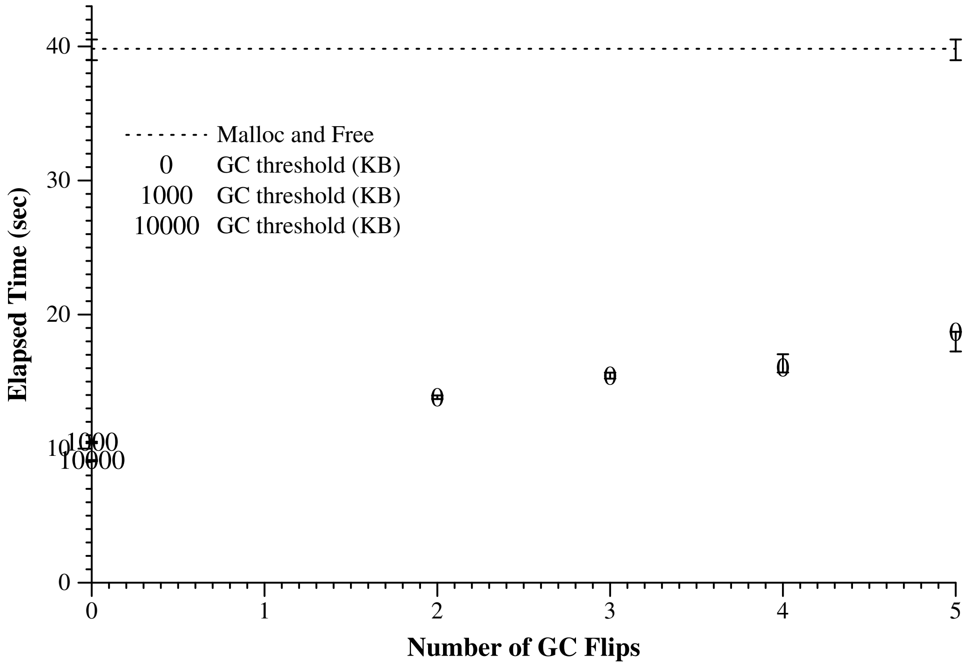


Figure 8.7: Sidney Times versus GC Flips (multiprocessor)

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
Trace Overhead		Sidney	5.7	100.0	-0.4	0.9
Trace Overhead	CPU	Sidney	5.7	100.0	-0.4	0.9
No GC		Sidney	9.1	100.0	-0.5	0.6
No GC	CPU	Sidney	8.3	100.0	-0.5	1.0

Table 8.8: Sidney Times (multiprocessor)

the cost to replay the trace is 10.2 sec elapsed (5.4 sec CPU). When no collection is done, the GC and I/O threads are inactive, which means that the difference between the elapsed and CPU times is the time doing I/O, 4.8 sec. The overhead to replay the trace is 3.1 sec. Subtracting the trace overhead from the cost when no collection is done gives the combined cost of `commit` and allocation, 7.1 sec elapsed (2.3 sec CPU); thus I/O takes 4.8 sec.

Figure 8.7 and Table 8.8 give the **coda** throughput results for the multiprocessor.

Even in the least favorable case, Sidney is more than 50% faster than RDS. The cost is linear in the number of collections and is 2.0 sec per GC. Collection is almost a factor of three cheaper than on the uniprocessor. The overhead to process the trace is 5.7 sec. When no collection occurs, the cost is 9.1 sec elapsed (8.3 sec CPU). From the last two results, we can conclude that the combined cost of `commit` and allocation is 3.4 sec elapsed (2.6 sec CPU) and thus I/O takes 0.8 sec. The I/O cost on the uniprocessor is more than a factor of five higher.

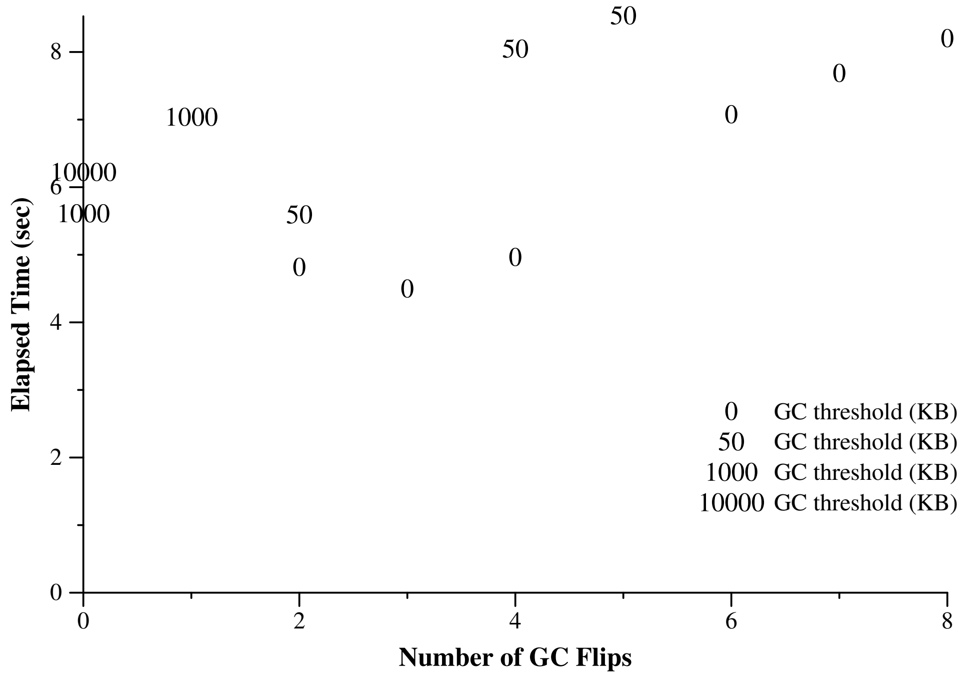


Figure 8.9: `Commit` Times versus GC Flips(uniprocessor)

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
Commit		Sidney	6.2	60.0	-1.8	1.9
Commit	CPU	Sidney	1.3	25.0	-2.3	2.3

Table 8.10: `Commit` Times, No GC (uniprocessor)

`Commit` Cost

This subsection examines the cost of `commit` for Sidney. For RDS, subtracting the trace overhead and the cost of `commit` allowed us to find the combined cost of `malloc` and `free`, but it was not possible to separate the two. For Sidney, the cost with no collections is known, so it is possible to find the cost of allocation alone. The only new measurement that is needed is the cost of `commit`. Thus not only is the `commit` cost inherently interesting, but measuring it also allows us to find the cost of allocation.

Figure 8.9 and Table 8.10 shows the cost of `commit` for the uniprocessor.

Since `commit` and GC are independent, one might expect the cost of `commit` to be independent of the number of collections. The figure shows that this expectation is not true; for a given GC threshold, the cost of `commit` is roughly linear in the number of flips. When the GC threshold is 50 KB, the slope is 1.0 sec per GC. I examined the CPU times for the data shown in Figure 8.9 and found that they do not depend on the number of flips. Since `commit` takes more elapsed time, but not more CPU time, either the I/O for `commit`

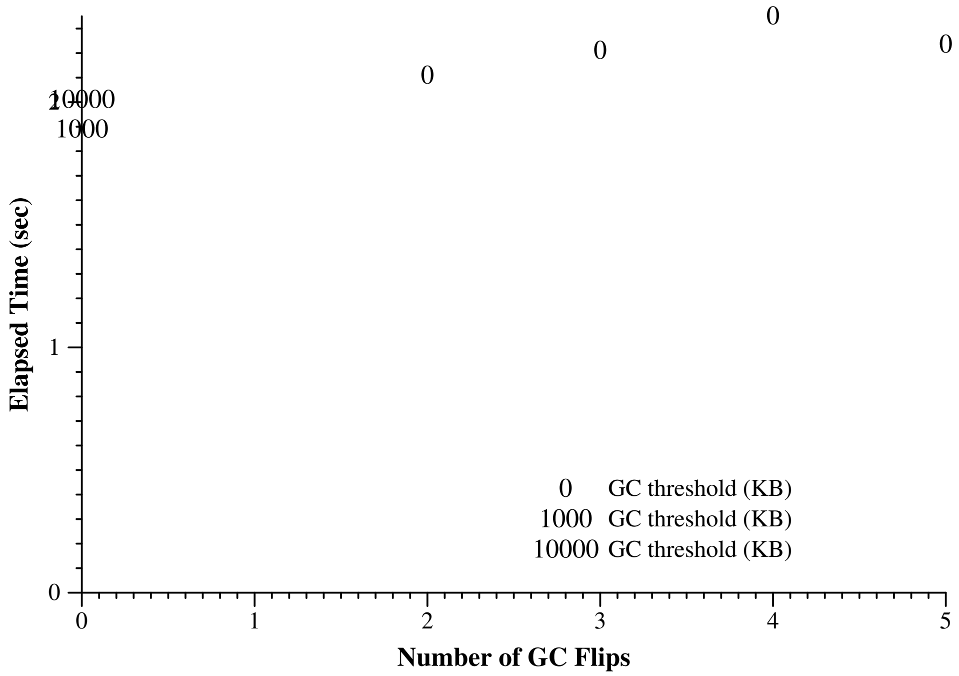


Figure 8.11: Commit Times versus GC Flips (multiprocessor)

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
Commit		Sidney	2.0	22.0	-1.3	3.1
Commit	CPU	Sidney	1.2	14.0	-1.7	2.5

Table 8.12: Commit Times, No GC (multiprocessor)

is taking longer or the slowdown is a result of having to share the CPU with the collector. It is less clear why there is a dependence on the GC threshold. When no collection occurs, the cost of `commit` is 6.2 sec elapsed (1.3 sec CPU). In comparison, the cost of `commit` for RDS is 43.2 sec elapsed (13.6 sec CPU). The large difference is especially notable since the frequency of commits was chosen to minimize the cost of RDS. Both systems have to move the same newly allocated data to disk, so the greatly increased `commit` costs almost certainly comes from the need to do RVM operations during each `malloc` and `free`. These operations add to the size of RVM data structures that `commit` must traverse, increasing CPU time, and to the amount of data that must be written to disk, increasing I/O time. From our earlier results for the combined cost of `commit` and allocation, 7.1 sec elapsed (1.3 sec CPU), we can conclude that the cost of allocation is 0.9 sec elapsed (1.0 sec CPU). In contrast, the elapsed time for RDS `malloc` and `free` is 23.6 sec. Even if the cost of `commit` were the same, Sidney would need both to allocate the data and to do almost four garbage collections to equal the cost of `malloc` and `free`.

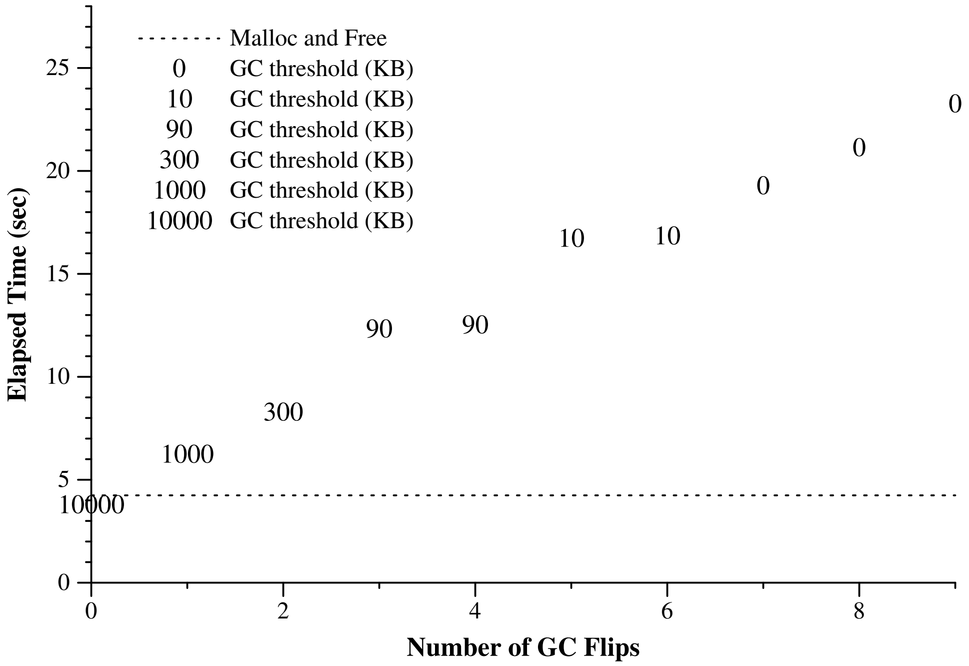


Figure 8.13: Sidney Times versus GC Flips, No RVM (uniprocessor)

Figure 8.11 and Table 8.12 shows the cost of `commit` for the multiprocessor case. The cost of `commit` is essentially constant, especially when compared to the uniprocessor case. Since the arrangement of the log and data segments on the disk is essentially the same on both machines, if the dependence of `commit` on the number of flips were due to I/O interference we would expect to see it here. Instead, the evidence supports the conclusion that the dependence is due to sharing the CPU; there is no interference in the multiprocessor case because the CPU is not shared. When no collection occurs, the cost of `commit` is 2.0 sec elapsed (1.2 sec CPU). This value compares to 16.3 sec elapsed (10.7 sec CPU) for RDS, again a significant difference. From our earlier results for the combined cost of `commit` and allocation, 3.4 sec elapsed (2.6 sec CPU), we find that allocation takes 1.4 sec of elapsed and CPU time. The cost of `malloc` and `free` is 19 sec. Sidney would need to allocate the data and to do almost nine collections to equal this value.

No RVM

In this part, I present the results of running Sidney with the RVM routines replaced with routines that just return and with all other persistence-related I/O turned off. This change allows us to separate the cost of copying the data from from-space to to-space from the cost of writing it to disk and using RVM to make the flip atomic. It gives an idea of how expensive persistence is, and helps us to understand whether Sidney outperforms RDS because of RVM-related overheads.

Figure 8.13 shows the dependence of Sidney's throughput on the GC threshold and the number of flips achieved by the collector when all RVM and I/O overhead has been eliminated on the uniprocessor. Table 8.14 gives more detailed information for the case in

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
No GC No RVM		Sidney	3.8	100.0	-0.1	0.8
No GC No RVM	CPU	Sidney	3.8	100.0	-0.4	0.4

Table 8.14: Sidney Times, No RVM (uniprocessor)

which the collector is never invoked.

Not surprisingly, eliminating RVM and the other I/O substantially improves the collector's performance. Processing the trace takes 3.8 sec when there are no garbage collections, as compared to 10.2 sec with RVM. The cost of `commit` is negligible, so subtracting the trace overhead, 3.1 sec, gives the time spent in allocation, 0.7 sec. This value is quite close to the result with RVM, 1.0 sec, which is to be expected since allocation does not do any I/O or RVM operations. The cost of collection is 2.2 sec per GC, as compared to 5.8 sec per GC with RVM. In both cases, the collectors must copy the same data; thus the difference between the two costs of collection gives the cost of the RVM and I/O portions of the collection, 3.6 sec per GC. The cost of RVM and writing the heap to disk is greater than to copy the data in memory. Finally, GC no longer outperforms malloc-and-free for most settings of the GC threshold. Since the cost of allocation is still less than that for malloc-and-free, there is some threshold where collection does better, but the length of the current trace is not great enough to measure where this occurs. This result shows that Sidney's advantage over RDS is almost entirely due to persistence-related overheads.

Figure 8.15 and Table 8.16 give the multiprocessor results when all RVM and I/O overhead has been eliminated.

With no collections, the total time is 7.2 sec, as compared to 9.1 sec with RVM. Subtracting the trace overhead, 5.7 sec, gives the cost of allocation, 1.5 sec. With RVM, the cost of allocation is 1.4 sec, which implies that the difference between the two cases is due to `commit`, not allocation. The higher cost of trace replay and allocation compared to the uniprocessor, may reflect differences in the cache structure of the two machines. The cost per GC is 1.6 sec as compared to 2.0 sec when RVM is used. The difference, 0.4 sec, is the time spent doing I/O and RVM operations. This difference is much smaller than for the uniprocessor, 3.6 sec, again reflecting the better I/O system of the multiprocessor. Malloc-and-free not only outperforms GC, but is even less expensive than allocation. However, since Sidney and RDS were not designed for volatile data, I would caution the reader not to draw any strong conclusions about non-persistent GC and malloc-and-free from this result. Also this experiment is not designed to make a careful comparison between GC and malloc-and-free for transitory data. The only conclusion to draw is just that Sidney's advantage over RDS is primarily due to RVM and I/O overheads. The ability to allocate from a single portion of free space, seems to be a significant advantage over using a free list as well.

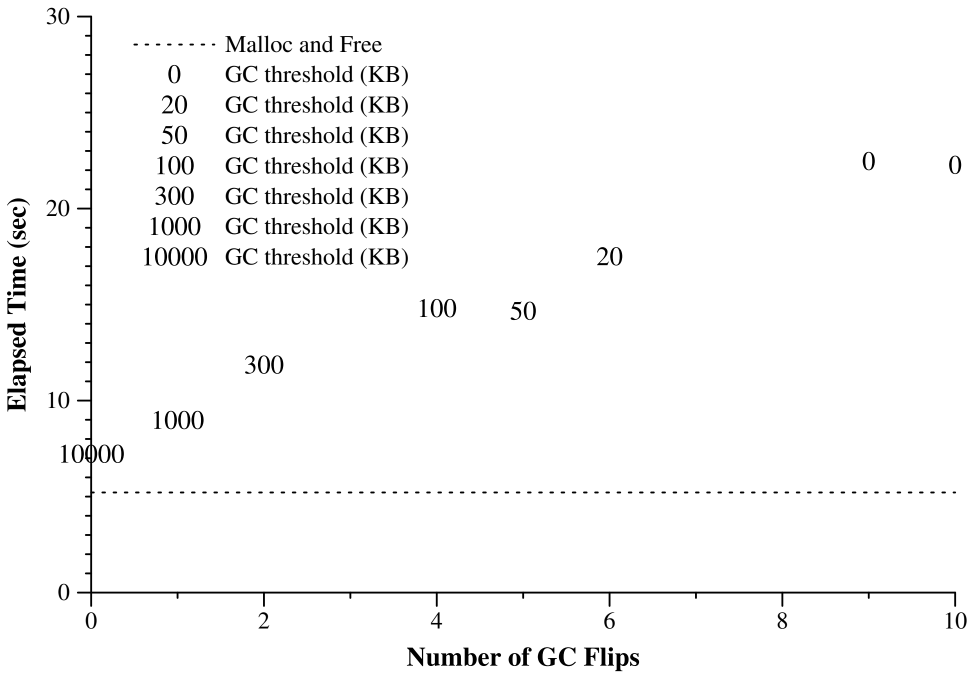


Figure 8.15: Sidney Times versus GC Flips, No RVM (multiprocessor)

Quantity Measured		Benchmark Used	Median (sec)	Total (%)	25th (%)	75th (%)
No GC No RVM		Sidney	7.2	100.0	-1.1	1.1
No GC No RVM	CPU	Sidney	7.2	100.0	-1.1	1.2

Table 8.16: Sidney Times, No RVM (multiprocessor)

8.2.4 Stop-and-Copy Throughput

Finally, it is interesting to look at the performance of stop-and-copy collection, to understand the cost of GC in situations where pause times are not critical and to understand better the advantages and disadvantages of Sidney. Since they are essentially the same as the uniprocessor results, I have omitted the multiprocessor results.

Figure 8.17 shows the basic results of replaying the trace on the uniprocessor in the usual format. In the interest of brevity, I've omitted the plots and tables for the other measurements discussed in the following two paragraphs.

The results are similar to those for Sidney, except that collections are more expensive since I/O cannot be overlapped with computation, and because collections are now deterministic, each GC threshold results in a fixed number of flips. A GC threshold of 0 KB results in 28 flips, compared to at most 8 for Sidney. Care should be taken when comparing plots: the greater number of flips makes the slope look much larger than it is. Without collections, the cost to replay the trace is 10.2 sec elapsed (5.4 sec CPU), exactly the same

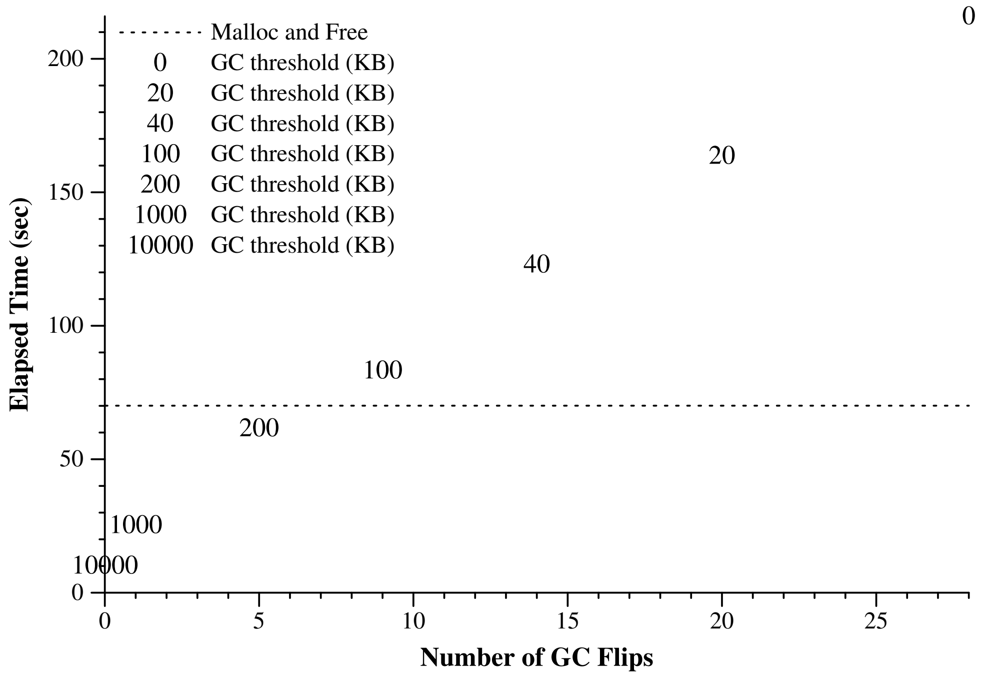


Figure 8.17: Stop-and-Copy Times versus GC Flips (uniprocessor)

as for Sidney. Since they are doing the same work and the GC and I/O threads are inactive, this result is expected. The cost per GC is 7.3 sec, compared to 5.8 sec for Sidney.

I also measured the cost of `commit`; it is not dependent on the GC threshold and has the same cost as Sidney in the no-collection case. I also measured the system with RVM and I/O turned off. In the no-collection case, the results were basically the same as above. The cost per GC is 2.0 sec. Thus the cost of RVM and I/O for stop-and-copy collection is 5.3 sec, in contrast to 3.6 sec for Sidney. Sidney must do the same RVM and I/O operations (and possibly more) as stop-and-copy. Thus the reduced cost of RVM and I/O for Sidney reflects Sidney's ability to overlap I/O with computation. Since the copying costs as reflected by the cost without RVM are the same, all of Sidney's advantage over stop-and-copy comes from this overlap, as is to be expected on a uniprocessor.

8.3 Summary

To demonstrate that implicit storage management techniques can be used in place of explicit ones for persistent storage, it is essential to compare persistent GC to persistent malloc-and-free. This comparison is difficult to make because it is not feasible to run the same persistent application using both GC and malloc-and-free. Instead, I traced the memory management operations of a realistic application, the Coda file system. Then, I replayed the traces using a program designed to be able to use both techniques, thus allowing me to make a direct comparison.

The results bear out the claims of the thesis. First, generating the trace took almost a month of Coda use, but replaying it takes only minutes. Thus persistent memory management has a negligible impact on Coda, and substituting GC for malloc-and-free, if it were possible, would not hurt Coda's throughput. Second, because GC has lower RVM-related overheads, it is actually faster than malloc-and-free, even for aggressive settings of the GC threshold. Finally, because it can overlap computation with I/O, Sidney is faster than stop-and-copy collection.

Chapter 9

The Comp Benchmark

Comp is the most real-world benchmark used in this dissertation. It demonstrates the impact of Sidney on the performance of a substantial and widely-used application, the SML/NJ compiler. Most importantly, **comp** is used to investigate the effect of concurrent collection on GC pause times; such an investigation is crucial to demonstrating the thesis. In addition, it is used to take a detailed look at the factors influencing throughput in Sidney. The transactions found in **comp** are long-running and require the allocation and modification of substantial amounts of data in each transaction. The long transactions in **comp** are a useful contrast to the short transactions found in **trans**. Finally, because it makes a large optimizing compiler persistent, **comp** also serves as a prototypical example of a persistent programming environment.

Comp is based on a version of the SML/NJ compiler adapted to make its state persistent. The benchmark uses this modified compiler to compile files taken from the compiler's implementation. After each file has been compiled, the state of the system is committed to the disk. Sidney collects the persistent heap concurrently with compilation. Unlike the two previous benchmarks, **trans** and **coda**, which focus on specific aspects of Sidney's performance, **comp** allows all aspects of Sidney's performance to be studied in some detail.

The results of running **comp** support the thesis. Chapter 6, showed that Sidney achieves modest speedups relative to stop-and-copy collection. In this chapter, we will investigate the source of this speedup and see that the speedup is about half of the maximum possible. More importantly, we also saw in Chapter 6 that the concurrent collector's pauses are several orders of magnitude shorter than the stop-and-copy collector's. In this chapter, we will study the contribution of various aspects of Sidney's implementation to these pauses and propose some ways to further improve pause times. In general, the more detailed performance analysis presented here allows a deeper understanding of the current system and provides insight that should enable future systems to improve on Sidney's performance.

The chapter begins with a discussion of the benchmark itself, including an explanation of how the compiler state was made persistent. The next section examines the performance of **comp**. It begins by looking at the throughput of the collector on the multiprocessor and the uniprocessor, and follows with an examination of the `commit` throughput on the multiprocessor. The final part of the performance section is an investigation of the latencies of GC and `commit`. The chapter concludes with a brief summary.

9.1 Description of Comp

My goal in designing the **comp** benchmark was to provide an example based on a substantial SML program that was in widespread production use. Both **trans** and **OO1** are “toy” benchmarks in the sense that they were written not to perform some real task, but rather purely to act as benchmarks. The Coda file system is itself a real program, but the driver used by **coda** to replay the Coda traces is not. The SML/NJ compiler does not suffer from any of these problems; it is both a real program and a program whose performance is important to a significant user community.

Modifying SML/NJ

Standard ML of New Jersey is a large optimizing compiler for Standard ML in daily production use by a significant user community worldwide [36]. The version (0.75) used here consists of more than 50,000 lines of SML divided among 230 files. It was chosen as a benchmark in part because it is one of the largest and most widely used SML programs in existence. It differs from many SML programs by making a greater use of assignment than is usual. This greater use of assignment will probably make Sidney more expensive than might generally be expected for most SML programs.

I modified SML/NJ to store its state in the persistent heap. The main challenge was to make certain that all the state needed by the compiler was persistent. No attempt was made to identify exactly which compiler data structures needed to be persistent. Instead, when the compiler completes the compilation of a file, the current continuation is captured. The continuation is then assigned to a value reachable from the persistent root, making the continuation persistent. Recall that the current continuation captures all the information needed to continue the computation; hence all the state that must be persistent to restart the compiler must be reachable from it. After the continuation is captured a `commit` is done, guaranteeing that the continuation and all the data reachable from it is stored on disk.

Capturing the current continuation in this way implements a persistent process model in which a process can be saved in the persistent heap and restarted in the event of a crash. The ease with which the compiler was modified is a direct consequence of using the orthogonal persistence model. Orthogonal persistence allows essentially all of the state that needs to be persistent to be found implicitly by Sidney. If persistence were determined explicitly, it would have been necessary to make much more extensive modifications. Every datum that needed to be persistent would have to be identified and allocated in the persistent heap. Such a change would have been extremely difficult for a large pre-existing program like SML/NJ. As it is, the changes are less than 100 lines of SML, are entirely confined to a single file, and took only a few hours to make. Furthermore, there is no danger of missing some datum that should have been persistent, only to discover it when the system fails to recover from a crash. Needless to say, unexpected failures of recovery are best avoided.

Benchmark Setup

The next issue is how Sidney's command line parameters are set when running **comp**. The less significant parameters are set as described in Section 6.2.1. The more significant parameters are the thresholds for triggering major, minor, and persistent collections. I experimented with various values for the minor collector and major collector thresholds, but I finally used 1 MB for each. Larger values did not change the running time significantly and the parameters selected resulted in a reasonably sized working set. After some experimentation, I set the threshold for persistent collection more aggressively at 100 KB. This setting means that the concurrent persistent collector is almost always running. Continuous collection will tend to emphasize collection overheads, making them easier to study. In practice, a larger setting would probably be used, reducing the overall cost of persistent GC. Lower persistent GC costs would also reduce the throughput-related advantages of concurrent GC, although of course the more important latency advantages would remain.

Running Characteristics

When **comp** is run, the modified compiler is used to compile 37 files from the SML/NJ source. After each file is compiled, the continuation is captured and a `commit` is done making the current state stable. Although when running the actual benchmark no recovery is done, during testing **comp** crashed (sometimes deliberately) and was able to recover successfully. While **comp** is running, the concurrent collector is typically able to achieve five flips on the uniprocessor and four flips on the multiprocessor. Since the overall running time is very sensitive to the number of flips, any runs that flipped a different number from the most typical are eliminated. The **coda** benchmark already serves to illustrate the effects of differing numbers of flips. The stop-and-copy collector is run so that it replays the flips of the concurrent collector, as discussed in Section 6.2.1.

It is useful to examine the amount of data manipulated by `commit` and persistent collection before presenting the actual running times. On the uniprocessor, for a representative run, the stop-and-copy collector copies a total of 2829 KB, a minimum of 482 KB, a maximum of 674 KB, and average of 566 KB. The stop-and-copy collector copies exactly the live data; thus the amount copied by the stop-and-copy collector also gives the livesize, which increases monotonically as the compiler executes. Similarly, the concurrent collector copies a total of 2862 KB, a minimum of 483 KB, a maximum of 677 KB, and an average of 572 KB. The concurrent collector copies somewhat more data than the stop-and-copy collector because it sometimes copies something that later becomes garbage. For these examples, the concurrent collector only copies about 1% more data than the stop-and-copy collector. The results for the multiprocessor are similar and so are not repeated here.

`Commit` is deterministic, so all variations produce the same results. There were 37 `commits`, one for each file. The maximum amount of data copied from the transitory heap into the persistent heap (and from there to disk) was 26 KB, the minimum 5 KB, the average 11 KB, and the total 406 KB. In contrast, each transaction in **trans** involves only 56 bytes. Although 406 KB of data is added to the heap by `commit`, the persistent heap only grows by about 200 KB, implying that about 200 KB of data becomes garbage during the run.

Evaluation of Design Alternatives

By comparing the preceding numbers for `commit` and persistent GC, some insight can be gained into the possible performance of two design alternatives. The first alternative is to implement `commit` by checkpointing the entire heap and the second is adding generational collection to Sidney.

I rejected the idea of checkpointing the entire heap on a `commit` in favor of the incremental approach used by Sidney, in part because checkpointing would almost certainly have poor performance on small transactions such as those in **trans**. It is interesting to estimate how well checkpointing would do for the larger transactions found here. The average size of the heap is 566 KB; so, on average, each checkpoint would need to transfer over half a megabyte to disk, more that 50 times the amount Sidney transfers. However, Sidney must also move its data from the transitory heap to the persistent one, as well as do various housekeeping work, while checkpoint can basically just write the heap to disk. We will see later that I/O dominates the cost of `commit`, so it is likely that checkpointing will have a significant performance disadvantage despite its greater simplicity.

An obvious way to attempt to improve Sidney's GC performance would be to make Sidney a generational collector. Nothing in Sidney's basic design prevents this, and in fact the transitory heap used by Sidney is generational. Any deep understanding of this enhancement would require a detailed design; here I only attempt a back-of-the-envelope analysis. Assume that when doing a `commit`, we move data into the first persistent generation, and that the five collections move any live data in the first generation into the next one. Then the amount of data copied into the first generation would be an upper bound on the amount of data the collections could copy. This upper bound is just the total copied by `commit`, or 406 KB, which is a factor of six less than the 2829 KB that the current collector copies. Even if one of the collections collected the entire heap, there would be a factor of three improvement. If the data that becomes garbage is young and thus in the first generation, then the improvements due to generational GC could be even more substantial because then the total amount of data that needed to be copied would be closer to 200 KB. Before actually adding generations to Sidney, it would be useful to test this hypothesis.

9.2 Results of Comp

This section presents a detailed analysis of the performance of **comp**, building on the high-level results presented in Chapter 6. The analysis covers four topics: the throughput of **comp** on the multiprocessor, the throughput of **comp** on the uniprocessor, the throughput of `commit`, and the latencies of GC and `commit`. In summary, the results are:

Multiprocessor Throughput

> Compared to stop-and-copy collection, using concurrent collection results in an overall speedup of 3%. About 40% of the speedup comes from overlapping I/O with computation. Measuring the speedup of collection alone shows that the maximum possible speedup is about twice what is actually achieved. The time spent in the run-time is 15-20% of the total running time, and the slowdown relative to the maximum

possible speedup is distributed proportionally between user and runtime code. It is not clear what the source of the slowdown is. The GC and I/O threads offload essentially all of the collection-related work from the client thread. The cost of writing to-space to disk is a factor of four greater than the cost of copying the data from from-space to to-space. The cost of truncation, which could be eliminated by modifying RVM, accounts for half of the cost of writing to-space.

Uniprocessor Throughput

Using concurrent collection results in an overall speedup of 8%. Since the machine is a uniprocessor, all of this speedup is due to overlapping I/O with computation. The much greater cost of I/O for this machine accounts for the greater throughput improvement compared to the multiprocessor. The speedup achieved is somewhat more than half of the maximum possible. The time spent in the runtime is 30-40% of the total elapsed time, and the runtime is responsible for more than 50% of the slowdown relative to the maximum possible speedup. Thus, the runtime bears a proportionally greater share of the slowdown than the user code. Again, the exact source of the slowdown is not clear. When concurrent collection is used, essentially all of the collection work found in the stop-and-copy case is transferred to the GC and I/O threads. However, since the same CPU is used to execute these threads, the CPU time for these threads contributes to the overall elapsed time. In fact, because the GC and I/O threads have greater CPU overheads than the stop-and-copy collector, some of the possible speedup is lost. The cost of writing to-space to disk is eight to nine times greater than the cost of copying the data from from-space to to-space. Again, truncation accounts for about half the I/O-related cost and could be eliminated.

`Commit` **Throughput**

The transactions done by **comp** involve 200 times more data than those done by **trans**, and are done 170 times less frequently. Despite this, each `commit` is only a factor of six more expensive for **comp**. The synchronous write of the RVM log is the largest contributor to this cost, taking more than twice the rotational latency of the disk. This value implies that, unlike for **trans**, `commit` spends a significant amount of time transferring data to the disk. The primary cost of orthogonal persistence, moving data from the transitory heap into the persistent heap, is only 0.2% of the overall cost of **comp**.

GC and `Commit` **Latencies**

The most important result of **comp** is to show that the pauses created by concurrent collection are not unacceptably long or too frequent. The longest pauses that occur with any significant frequency are less than or equal to 64 msec long, a factor of 30 shorter than the shortest pauses caused by stop-and-copy collection. Furthermore, the longest collector pauses are no longer than the shortest `commit` pauses and much less frequent. Thus, the user creates pauses that are more disruptive than those created by Sidney. The major contributor to the collector pause time is the routine that redirects pointers from the transitory heap into the persistent from-space to the persistent to-space when the collector flips. This routine is currently implemented in a rather brute-force way and is a likely candidate for optimization.

9.2.1 Comp Throughput

This subsection presents the throughput-related measurements for **comp**, both recapitulating the high-level results presented in Chapter 6 and presenting additional measurements of the components of Sidney that influence those results. In repeating the high-level results, I have included the values of the quartiles and measurements of the CPU times. The quartile values show the shape of the distributions and bound the error in the median. The CPU times allow us to understand the effects of I/O and concurrency on **comp**.

In presenting the GC results, for each processor I first repeat the high-level results as well as present the costs of persistent GC. Then I present a breakdown of the costs of the client thread when executing in the runtime. Finally, I present a breakdown of the costs for the GC and I/O threads.

The results for `commit` begin with a detailed breakdown of the cost of `commit` for **comp**. The same breakdown of `commit` costs is then presented for **trans**, repeating some measurements from Chapter 7, to allow the two cases to be easily compared.

Multiprocessor Throughput

The total time to run **comp** is the most important measure of how successful Sidney is at improving its throughput. The next most significant measurement is the time spent executing the garbage collector in the client thread. Sidney only makes collection concurrent, so the difference between the time spent in the client thread in the stop-and-copy case and the concurrent case bounds the amount of speedup that might be achieved overall. For the multiprocessor, since all of the persistent GC work is transferred to other threads running on separate processors, any difference between this bound and the speedup actually achieved is due to overheads created by concurrent collection.

Table 9.1 contains the results of running **comp** for both total time and persistent GC. It has the standard format described in Chapter 6. The rows marked *Total Time* are the times for running the entire **comp** benchmark (CPU time is for the client thread only) and the rows marked *Persistent GC* are the times for the persistent collector running in the client thread.

When concurrent collection is used, the total time shows a speedup of 2.5 sec elapsed (1.6 sec CPU), which is 3% of the total elapsed time for stop-and-copy. The difference between the elapsed time and CPU time speedups, 0.9 sec, is due to overlapping I/O with computation, while the CPU speedup is due to transferring work to the other threads executing on separate processors. Executing the collector, stop-and-copy collection spends 4.2 sec elapsed (2.9 sec CPU) and concurrent collection spends 0.2 sec. Thus, the maximum possible speedup would be 4.0 sec elapsed (2.7 sec CPU). In practice the system achieves a little more than half the possible speedup.

It is useful to comment on the distributions. For the larger median values, such as total time, the distributions are quite tight with a spread of a few percent. The smallest

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Total Time		Concurrent	77.3	100.0	-0.7	1.6
Total Time	CPU	Concurrent	76.0	100.0	-0.6	0.9
Total Time		Stop&Copy	79.7	100.0	-0.4	0.4
Total Time	CPU	Stop&Copy	77.6	100.0	-1.0	0.2
Persistent GC		Concurrent	0.2	0.3	-4.5	7.4
Persistent GC	CPU	Concurrent	0.2	0.3	-4.6	14.0
Persistent GC		Stop&Copy	4.2	5.3	-0.7	1.6
Persistent GC	CPU	Stop&Copy	2.9	3.7	-0.7	2.1

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Total Time	-2.5	-1.6	1.3	2.1
C Persistent GC	-4.0	-2.7	0.0	1.3

Table 9.1: Total and Persistent GC Times (multiprocessor)

values, such as for concurrent GC, show a greater spread, reflecting the greater difficulty of measuring these smaller quantities accurately. For the smaller values, we also see that the CPU measurements have a larger spread, reflecting the reduced precision of the CPU timer. In general, the stop-and-copy results have slightly smaller spreads, probably because of the non-determinism inherent in concurrent collection. Generally, the 75th percentile values are further from the median than the 25th percentiles. This difference reflects the fact that the distributions tend to be skewed towards larger values, as seen in Chapter 6. None of the measurements show enough spread in the distribution to call into question the basic results. Since these observations generally hold for the results in this chapter, unless there is a significant exception, I will omit further discussion of the distributions.

Runtime Overheads

Why is only half of the possible speedup observed? The time taken by the client thread determines the elapsed time of the program, so in this part I attempt to answer this question by looking at the costs of the client thread. I begin by first looking at the overall time spent in the runtime system; the runtime is especially relevant because it contains almost all of Sidney's implementation. The difference between the total time and the time in the runtime is the time for the user code, which is the other part of the client thread costs. For the runtime system, it is also possible to measure the time spent in individual components. The key components of the runtime overhead are the costs of `commit`, and minor, major, and persistent garbage collection. Measuring these quantities allows us to understand both the basic overheads of the system, which is useful in itself, and their contributions to the slowdown relative to persistent GC.

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Runtime		Concurrent	16.4	21.0	-1.3	1.2
Runtime	CPU	Concurrent	15.6	21.0	-1.1	1.9
Runtime		Stop&Copy	20.1	25.0	-0.7	1.4
Runtime	CPU	Stop&Copy	18.0	23.0	-1.0	0.6
Minor GC		Concurrent	13.3	17.0	-0.3	0.6
Minor GC	CPU	Concurrent	13.3	17.0	-0.8	0.8
Minor GC		Stop&Copy	13.3	16.0	-0.7	0.4
Minor GC	CPU	Stop&Copy	13.3	17.0	-0.7	0.3
Commit		Concurrent	2.8	3.6	-6.6	1.8
Commit	CPU	Concurrent	1.8	2.4	-2.2	2.7
Commit		Stop&Copy	2.6	3.3	-8.1	5.2
Commit	CPU	Stop&Copy	1.8	2.3	-3.4	5.7
Major GC		Concurrent	1.0	1.2	-1.9	3.5
Major GC	CPU	Concurrent	1.0	1.3	-4.1	2.0
Major GC		Stop&Copy	1.0	1.2	-2.0	4.1
Major GC	CPU	Stop&Copy	1.0	1.3	-3.1	6.1

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Runtime	-3.7	-2.4	0.9	2.1
Minor GC	0.0	0.0	0.0	0.0
Commit	0.2	0.0	1.0	0.9
Major GC	0.0	0.0	0.0	0.0

Table 9.2: Runtime Overheads (multiprocessor)

Table 9.2 shows the results of measuring the runtime and its components. The measurements are presented in roughly decreasing order of elapsed time. The rows labeled *Runtime* are for the entire runtime, those labeled *Minor GC* are for all of the minor collections, those labeled *Commit* are for `commit` and, those labeled *Major GC* are for the major collections. The measurements for persistent collection are not repeated.

The runtime system shows a speedup of 3.7 sec elapsed (2.4 sec CPU). This speedup represents a 0.3 sec slowdown relative to persistent GC. Subtracting the time spent in the runtime from the total time gives the time spent in user code. User code takes 59.6 sec when using stop-and-copy collection and 60.9 sec elapsed (60.4 sec CPU) when using concurrent collection. When concurrent collection is used, the user code slows down by 1.3 sec elapsed (0.8 sec CPU). The system spends three to four times as much time in user code as in the runtime. Thus, it is no surprise that the slowdown is also a factor of three to four greater for user code. I do not have a good explanation for why both the user code and the runtime show some CPU time slowdowns. I would expect any interference due to concurrent collection to change the elapsed times, but not to increase the CPU times. Some possible explanations might be that the system is not maintaining the CPU timers completely correctly, especially when there are threads, or that the presence of threads is causing some cache or bus interference that is reflected in the CPU times. Unfortunately, I have no data to support these suppositions.

Looking at the components of the runtime, we see that minor and major collection show no difference between collectors. The greater frequency of minor collections account for their significant expense compared to major collections. `Commit` shows a slowdown of 0.2 sec elapsed but no change in the CPU time, suggesting that there is some I/O-related interference. Adding the contributions of major and minor collection and `commit`, we would expect the slowdown in the runtime to be 0.2 sec elapsed (0.0 sec CPU). These values roughly agree with the 0.3 sec elapsed calculated previously, but do not confirm the 0.3 sec CPU also calculated previously. I suspect this inconsistency is due to errors in the measurements that have been emphasized by subtracting and adding a number of quantities of similar magnitude. It may also be that the error lies in the initial estimate of the runtime slowdown; if so, it helps explain the puzzling CPU time changes.

Persistent GC Overheads

Next, consider the overheads associated with persistent collection. Broadly speaking the cost of collection is divided into two parts: the cost to copy and scan the data, and the cost to write it to disk and perform other actions relating to persistence. For the stop-and-copy collector, all of these are part of the client thread, but for the concurrent collector, they are mostly borne by the GC and I/O threads. The previous section has already shown that the contribution of collection to the client thread's running time is small.

I examine the copying-related costs first. For the concurrent collector, essentially all of the copying is done by the GC thread; thus the copying on the multiprocessor should not affect the elapsed time of the overall process. Copying is the only important component of the cost of the GC thread.

Unlike copying and scanning, the costs of writing the to-space are unique to persistent collection. The main operations are writing the heap to disk and truncating the RVM log. Roughly the same amount of data must be written to disk as was copied. It is likely that writing the data to disk is more expensive than copying it in memory, so we would expect these costs to dominate those of copying. It is more difficult to speculate about the cost of truncation, but it is useful to remember that this cost could be avoided by making conceptually simple modifications to RVM as discussed in Chapter 5.

Table 9.3 shows the results of measuring the costs of copying and writing to-space for the multiprocessor. The rows marked *GC Copying* are for copying and scanning the data. The rows marked *To-Space I/O* are for all of the I/O-related activities involved in writing to-space, those marked *Truncate* are just the cost of truncating the RVM log, and those marked *Write Heap* are for writing the heap.

Concurrent collection appears to use more CPU time than elapsed time, which is impossible. This discrepancy indicates that the last digit of the CPU time is not entirely reliable, which would also help explain some of the problems in the previous parts. The

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
GC Copying		Concurrent	1.2	1.5	-1.8	2.7
GC Copying	CPU	Concurrent	1.3	1.7	-14.0	20.0
GC Copying		Stop&Copy	0.9	1.1	-0.5	0.5
GC Copying	CPU	Stop&Copy	0.9	1.1	-1.1	1.1
To-Space I/O		Concurrent	4.9	6.1	-2.9	4.8
To-Space I/O	CPU	Concurrent	3.0	4.0	-2.7	6.1
To-Space I/O		Stop&Copy	3.2	3.9	-0.5	1.7
To-Space I/O	CPU	Stop&Copy	1.8	2.3	-0.6	1.7
Truncate		Concurrent	2.4	3.1	-4.7	30.0
Truncate	CPU	Concurrent	1.3	1.8	-6.0	75.0
Truncate		Stop&Copy	1.6	2.0	-2.3	3.9
Truncate	CPU	Stop&Copy	0.7	0.9	-2.8	4.2
Write Heap		Concurrent	2.4	3.1	-9.3	9.0
Write Heap	CPU	Concurrent	1.9	2.5	-7.9	7.9
Write Heap		Stop&Copy	1.6	2.0	-0.5	1.0
Write Heap	CPU	Stop&Copy	1.1	1.4	-1.9	1.9

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
GC Copying	0.3	0.4	-0.1	0.0
To-Space I/O	1.7	1.2	1.9	1.4
Truncate	0.8	0.6	1.1	0.9
Write Heap	0.8	0.8	0.5	0.5

Table 9.3: GC Overheads (multiprocessor)

large spread seen in the quartiles for the CPU time support this explanation. The concurrent collector spends 0.3 sec elapsed (0.4 sec CPU) more time than the stop-and-copy collector copying data. This longer time reflects the higher overheads in the concurrent GC thread caused by doing the copying in an incremental fashion. Since concurrent collection copies only 1% more data, that is probably not the source of the difference. The transfer of work from the stop-and-copy collection to the GC thread accounts for 0.9 sec of speedup for the collector.

The concurrent collector spends 1.7 sec elapsed (1.2 sec CPU) more time writing to-space than the stop-and-copy collector, a significant difference. This difference is probably because the concurrent collector writes the data incrementally, leading to higher overheads. Also, the concurrent collector may write the data more than once, since it is possible that the to-space data will change to reflect an assignment to the from-space data. The costs for writing the heap and for truncation show the same trends. Transferring the I/O-related work to the I/O thread accounts for 3.2 sec elapsed (1.8 sec CPU) of savings to the client

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Total Time		Concurrent	70.4	100.0	-0.3	0.5
Total Time	CPU	Concurrent	62.8	100.0	-0.2	0.2
Total Time		Stop&Copy	76.5	100.0	-0.4	0.3
Total Time	CPU	Stop&Copy	62.9	100.0	-0.2	0.2
Persistent GC		Concurrent	0.2	0.3	-4.4	9.4
Persistent GC	CPU	Concurrent	0.2	0.4	-6.7	13.0
Persistent GC		Stop&Copy	13.3	17.0	-1.3	1.0
Persistent GC	CPU	Stop&Copy	2.8	4.4	-1.1	1.7

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Total Time	-6.1	-0.2	7.6	14.0
Persistent GC	-13.1	-2.6	0.0	10.5

Table 9.4: Total and Persistent GC Times (uniprocessor)

thread. Combining this savings with the savings for copying, 0.9 sec, gives a savings for persistent GC of 4.1 sec elapsed (2.7 sec CPU), matching the measured value of 4.1 sec elapsed (2.7 sec CPU). Finally, even for stop-and-copy collection, the cost of writing the heap is three times more expensive than copying the data. If we eliminate the cost of truncation, the elapsed time cost difference is still almost a factor of two.

Uniprocessor Throughput

Again, the most important measurements are of the total running time and of the time spent by collection as part of the client thread. However, there are two important differences between the uniprocessor and the multiprocessor. First, because all the threads share a single processor, any apparent savings of CPU time for one thread will only be transferred to another thread, since none of the techniques used in Sidney actually reduces the amount of work to be done. This shift is likely to result in more overhead rather than less since there will be added overheads from threads. Second, the cost of I/O is much greater on the uniprocessor, which is a function primarily of file system differences, not the number of processors. Since Sidney achieves some of its speedups by overlapping I/O with computation, the greater I/O costs provide a greater opportunity for improvement. In fact, despite that on the uniprocessor the only opportunity for speedup is I/O overlap, the uniprocessor still has considerably greater throughput improvements than the multiprocessor.

Table 9.4 shows the results for both total time and for persistent garbage collection for the uniprocessor and corresponds to Table 9.1 for the multiprocessor.

When concurrent collection is used, the total time shows a speedup of 6.1 sec elapsed (0.2 sec CPU), which is 8% of the total elapsed time for stop-and-copy, and thus more than

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Runtime		Concurrent	21.0	30.0	-0.8	1.4
Runtime	CPU	Concurrent	17.0	27.0	-1.3	0.9
Runtime		Stop&Copy	30.5	40.0	0.0	0.6
Runtime	CPU	Stop&Copy	18.8	30.0	-0.8	0.7
Minor GC		Concurrent	14.6	21.0	-1.1	0.4
Minor GC	CPU	Concurrent	14.2	23.0	-0.7	0.9
Minor GC		Stop&Copy	13.9	18.0	-0.2	0.2
Minor GC	CPU	Stop&Copy	13.9	22.0	-1.0	0.9
Commit		Concurrent	4.4	6.3	-6.4	2.9
Commit	CPU	Concurrent	2.2	3.5	-2.8	3.5
Commit		Stop&Copy	3.1	4.1	-2.1	2.8
Commit	CPU	Stop&Copy	1.8	2.8	-3.5	2.6
Major GC		Concurrent	1.1	1.6	-2.0	3.1
Major GC	CPU	Concurrent	1.1	1.8	-2.1	6.2
Major GC		Stop&Copy	1.2	1.5	-4.0	7.8
Major GC	CPU	Stop&Copy	1.2	1.8	-2.7	5.4

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Runtime	-9.5	-1.8	4.0	12.0
Minor GC	0.6	0.3	0.3	0.0
Commit	1.3	0.4	2.2	1.3
Major GC	0.0	0.0	0.0	0.0

Table 9.5: Runtime Overheads (uniprocessor)

twice the relative speedup for the multiprocessor (3%). The greater overall speedup for the uniprocessor reflects the greater cost of I/O on the uniprocessor. This cost can be seen in the last two numbers of the summary lines, which give the differences between the elapsed and CPU times. These values are (relatively) much larger than on the multiprocessor, indicating a much greater cost of I/O. Persistent GC shows an improvement of 13.1 sec elapsed (2.6 sec CPU). Since the CPU "savings" will just be transferred to the other threads, it must be subtracted from the elapsed time saving to get the maximum elapsed time speedup, 10.5 sec. Thus, the system as a whole achieves about 65% of the maximum possible speedup, somewhat better than for the multiprocessor.

Runtime Overheads

To explain why the full speedup is not observed, and in general, to gain a better understating of the system, we look at the costs of the client thread. The same measurements that were presented for the multiprocessor are presented here for the uniprocessor.

Table 9.5 shows the detailed measurements of the runtime system for the uniprocessor and corresponds to Table 9.2 for the multiprocessor case.

The runtime system shows a speedup of 9.5 sec elapsed (1.8 sec CPU), which represents a slowdown relative to the persistent collector of 3.6 sec elapsed (0.8 sec CPU). Since the elapsed time will also be affected by the GC and I/O threads' use of the CPU, in the concurrent collector case, the elapsed time for the client code cannot be found just by subtracting the runtime overhead from the total time. However since the CPU clock is maintained on a per thread basis, we can find the user code CPU time this way. User code takes 46.0 sec elapsed (44.1 sec CPU) when using stop-and-copy collection, and 45.8 sec CPU when using concurrent collection. Thus, user code shows a slowdown of 1.7 sec CPU when concurrent collection is used. I will estimate the elapsed time slowdown for user code when the GC and I/O thread results are discussed.

Minor collection is again the biggest component of the runtime cost, accounting for 0.6 sec elapsed (0.3 sec CPU) of the runtime slowdown. Since minor collection does no I/O, the difference between the elapsed and CPU times are caused by sharing the CPU. As for the multiprocessor, the increase in CPU time is puzzling. The cost of major collections is small and essentially constant; the summary lines show that the differences are less than 0.05 sec, even though as printed, 1.2 sec versus 1.1 sec, they appear to be 0.1 sec. `Commit` makes the largest contribution to the runtime slowdown: 1.3 sec elapsed (0.4 sec CPU). It is possible that the entries added to the RVM log by concurrent collection help to account for the CPU time increase. Taken together, the components of the runtime account for a total slowdown of 1.9 sec elapsed (0.7 sec CPU) relative to persistent GC. These values match the observed value of 0.8 sec CPU, but vary a good deal from the observed elapsed time value of 3.6 sec elapsed.

The previous discrepancy seems large enough to merit investigation. On both the multiprocessor and the uniprocessor, summing the individual contributions to the runtime overhead results in a value that is about one second longer than the measured overhead. There is one exception: the elapsed time for the concurrent collector running on the uniprocessor, which is almost a second shorter. This observation explains why there is the large discrepancy, but it does not explain why the sums are too high or too low. It is not surprising that some of the numbers are too high; if each measurement has a small extra overhead, adding a number of them will amplify the error. I am unable to explain why concurrent GC on the uniprocessor is low.

Persistent GC Overheads

Again, I look at the costs of copying the data to to-space and of the persistence-related overheads such as copying to-space to disk.

Table 9.6 shows the results of measuring the costs of the GC and I/O threads for the uniprocessor; it corresponds to Table 9.3 for the multiprocessor case.

The concurrent collector spends 0.5 sec elapsed (0.2 sec CPU) more than the stop-and-copy collector copying and scanning data. The increase in CPU time is probably because the work is done incrementally. The GC thread does no I/O, so the difference of

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
GC Copying		Concurrent	1.8	2.6	-5.7	4.0
GC Copying	CPU	Concurrent	1.5	2.4	-3.1	1.0
GC Copying		Stop&Copy	1.3	1.7	-0.2	1.1
GC Copying	CPU	Stop&Copy	1.3	2.0	0.0	1.2
To-Space I/O		Concurrent	14.9	21.0	-1.1	0.5
To-Space I/O	CPU	Concurrent	1.6	2.5	-4.0	4.9
To-Space I/O		Stop&Copy	11.8	15.0	-1.1	1.9
To-Space I/O	CPU	Stop&Copy	1.3	2.1	-4.8	1.2
Truncate		Concurrent	8.3	12.0	-1.0	3.3
Truncate	CPU	Concurrent	0.7	1.1	-4.7	4.7
Truncate		Stop&Copy	5.7	7.4	-1.0	0.9
Truncate	CPU	Stop&Copy	0.6	0.9	-8.1	11.0
Write Heap		Concurrent	6.4	9.1	-3.4	3.9
Write Heap	CPU	Concurrent	0.8	1.3	-1.9	3.8
Write Heap		Stop&Copy	6.3	8.2	-5.1	1.6
Write Heap	CPU	Stop&Copy	0.7	1.1	-2.2	2.2

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
GC Copying	0.5	0.2	0.3	0.0
To-Space I/O	3.1	0.3	13.3	10.5
Truncate	2.6	0.1	7.6	5.1
Write Heap	0.2	0.1	5.6	5.5

Table 9.6: GC Overheads (uniprocessor)

0.3 sec between its elapsed and CPU times comes from sharing the CPU; together with the increased CPU time, this difference accounts for all of the elapsed time difference between the concurrent and stop-and-copy collectors.

The concurrent collector spends 3.1 sec elapsed (0.3 sec CPU) more than the stop-and-copy collector writing to-space. The increased CPU time probably comes from extra work needed to write the heap incrementally, while the elapsed time difference is probably due to sharing the CPU. Truncation takes 2.6 sec elapsed (0.1 sec CPU) more using concurrent collection, and writing the heap takes 0.2 sec elapsed (0.1 sec CPU) more.

The CPU times for the GC and I/O threads, 1.5 sec and 1.6 sec respectively, contribute to the overall running time. Adding these times to the elapsed time for the runtime, 21.0 sec, gives an estimate of 24.1 sec for the elapsed time excluding the user code. Subtracting this number from the total elapsed time, 70.4 sec, gives an estimate of 46.3 sec for the elapsed time of the user code. The estimated elapsed time for the user code when stop-and-copy collection is used is 46.0 sec, so there appears to be minimal slowdown of the user code.

Quantity Measured		Collector Used	Median (msec)	Total (%)	25th (%)	75th (%)
Commit		Stop&Copy	67.2	3.3	-8.1	5.2
Commit	CPU	Stop&Copy	45.1	2.3	-3.4	5.7
Synchronous Write		Stop&Copy	24.4	1.2	-4.6	3.4
Synchronous Write	CPU	Stop&Copy	1.8	0.0	-29.0	29.0
Minor Collection		Stop&Copy	20.9	1.0	-4.1	1.8
RVM Commit		Stop&Copy	6.2	0.3	-4.8	4.5
Log Writes		Stop&Copy	5.1	0.2	-1.5	1.3
Commit GC		Stop&Copy	5.0	0.2	-0.9	1.0
Pointer Roll-Back		Stop&Copy	4.7	0.2	-15.0	8.9
Free Pointer		Stop&Copy	3.0	0.2	-1.5	1.7

Table 9.7: **Comp** Commit Overheads (multiprocessor)

`Commit` Throughput

In this part, I look at the cost of `commit` in detail. I repeat the measurements of the overall cost of `commit`, but present them per `commit` to facilitate the later comparison to **trans**. New measurements examine the components of `commit`. These components include the synchronous RVM log write, the minor collection needed by `commit`, the commit GC, the logging of assignments to RVM, updating the persistent free pointer, the cost of RVM `end_transaction`, and the cost of rolling back pointers after a `commit`. Only the costs for the multiprocessor stop-and-copy case are considered here; the other cases are similar.

Table 9.7 shows the time taken for `commit` and its key subcomponents. It differs from the standard format in several important ways:

- The times presented are per `commit`, rather than for the entire run.
- The times are in milliseconds rather than seconds.
- Except where noted, the CPU times are essentially identical to the elapsed times and have been omitted.
- Since there are no concurrent GC results, the summaries have been omitted.

The table is organized in decreasing order of elapsed time. The rows labeled *Commit* are for the entire `commit` operation and those labeled *Synchronous Write* are for the synchronous write of the RVM log. The row labeled *Minor Collection* is for the minor collection done as part of `commit`; *RVM Commit*, for RVM `end_transaction` but excludes the log write; *Log Writes*, for logging the assignments on the store list to RVM; *Commit GC*, for the collection that moves data from the transitory heap to the persistent heap; *Pointer Rollback*, for the time spent doing pointer roll-back; and *Free Pointer*, for the time logging the new persistent data to RVM.

Quantity Measured		Benchmark Used	Median (msec)	Total (%)	25th (%)	75th (%)
Commit		Improved	11.1	96.0	-0.2	0.3
Commit	CPU	Improved	5.1	94.0	-3.6	3.6
Synchronous Write		Improved	5.9	51.0	-0.6	1.0
Synchronous Write	CPU	Improved	1.0	18.0	-4.1	12.0
Free Pointer		Improved	2.5	21.0	-1.4	0.9
RVM Commit		Improved	1.2	11.0	-3.8	3.8
Log Writes		Improved	0.6	5.3	-1.1	3.2
Minor Collection		Improved	0.4	3.8	-1.4	4.3
Commit GC		Improved	0.3	2.9	-1.7	2.4

Table 9.8: **Trans** Commit Overheads (multiprocessor)

Overall each `commit` takes 67.2 msec elapsed (45.1 msec CPU) and thus spends 22.1 msec doing I/O. As for **trans**, the synchronous RVM log write is the dominant individual factor taking 24.4 msec elapsed (1.8 msec CPU) and thus 22.6 msec I/O. The synchronous write accounts for 36% of the elapsed time and all of the I/O time for `commit`. The next most expensive component is the minor collection, 21 msec; this operation could be eliminated at the cost of making the commit GC slightly more complicated. Eliminating it would make `commit` faster, but since these garbage collections would need to be done eventually, it might not affect the overall running time. On the other hand, doing a collection early may force data to be copied that might have been dead if the collection is done later. Furthermore, if the minor collection can be done concurrently, it is likely to be better to do it out-of-line. The other factors are much less significant than those already discussed. The operations labeled by *RVM Commit*, *Log Writes*, and *Free Pointer* are all RVM intensive; together they account for 14.3 msec or 21% of the cost of `commit`. The commit GC, 5.0 msec, is the main overhead of orthogonal persistence and represents only 7% of the cost of `commit` and only 0.2% of the total cost of **comp**. Similarly, the cost of rolling back the transitory heap pointers is relatively small, 4.7 msec.

Comparison to Trans

The **trans** benchmark is also used to study the performance of `commit` in detail. **Trans** performs small transactions, transferring 56 bytes of data per transaction at a rate of about 5000 transactions per minute. In contrast, **comp** performs larger transactions, transferring on average 11 KB per transaction at a rate of about 30 transactions per minute. These differences are very significant; **comp** transfers 200 times more data per transaction and **trans** commits 170 times more frequently. Since the **trans** results that follow have been fully discussed in Chapter 7, all that appears here is a comparison to **comp**.

Table 9.8 repeats the information on the cost of `commit` for **trans**; it duplicates the Table 7.5 in Chapter 7 but with the addition of the rows *Synchronous Write*, which are taken from Table 7.3 in the same chapter.

For **comp**, the elapsed time for `commit` is six times greater than for **trans**. The CPU time is greater by a factor of nine. While these are significant increases, they are much less than the factor of 200 in the amount of data being committed. This observation suggests that the performance of **trans** is dominated by fixed overheads, while **comp** is more influenced by the actual size of the transactions. If **comp** were just doing `commit` and not actually compiling the files, it would perform about 15 TPS. Writing the RVM log takes a factor of four more elapsed time, but only a factor of two more CPU time. Since the elapsed time is much greater than the disk's rotational latency, 11 msec, at least part of that time is spent transferring data. The performance of **trans** is completely dominated by the rotational latency. The other components involve no I/O; of these, only writing the free pointer does not show a large increase in time. The increases range from a factor of 5 for RVM commit to a factor of 50 for the commit minor collection. The small change in the cost for writing the free pointer reflects that most of the work in this operation is independent of the amount of data involved in the transaction. The particularly large difference in the minor collection times arises because **comp** has much more live data that needs to be copied than **trans**.

9.2.2 Comp Latencies

Increasing throughput is a useful result of concurrent collection, but the main motivation for using it is to reduce GC pause times. In Chapter 6 we have already seen that the pauses caused by concurrent collection are an order of magnitude shorter than for stop-and-copy collection and that they are less disruptive than the `commit` pauses. Here, I investigate the factors that cause the pauses to be as great as they are. The goal is both to understand the system better and to identify ways of further improving the performance. This subsection presents these measurements for the uniprocessor only; the multiprocessor results are similar.

This section serves to highlight the differences that arise when attempting to optimize the system for throughput and latency. For stop-and-copy collection, the collector pauses examined here make up about 13% of the total running time, and for concurrent collection, only 0.2%. Reducing the stop-and-copy pause times by more than an order of magnitude results in a speedup of only 8%. For some applications, the concurrent collector pauses are still too long; the optimizations proposed below may well make it feasible to use GC in these applications, but will have a negligible effect of the running time.

Persistent GC and `Commit` pauses

In Chapter 6, I presented measurements of the GC and `commit` pauses for **comp** running on the multiprocessor. Here I present the same measurements for the uniprocessor. The two results are quite similar.

Figure 9.9 has the same form as Figure 6.6 from Chapter 6. For a detailed description see that figure. Briefly, note that the x-axis is logarithmic and gives the pause duration in milliseconds, while the y-axis is the number of pauses. The top plot shows the GC pauses, while the bottom one shows the commit pauses. Pauses from the system running concurrent

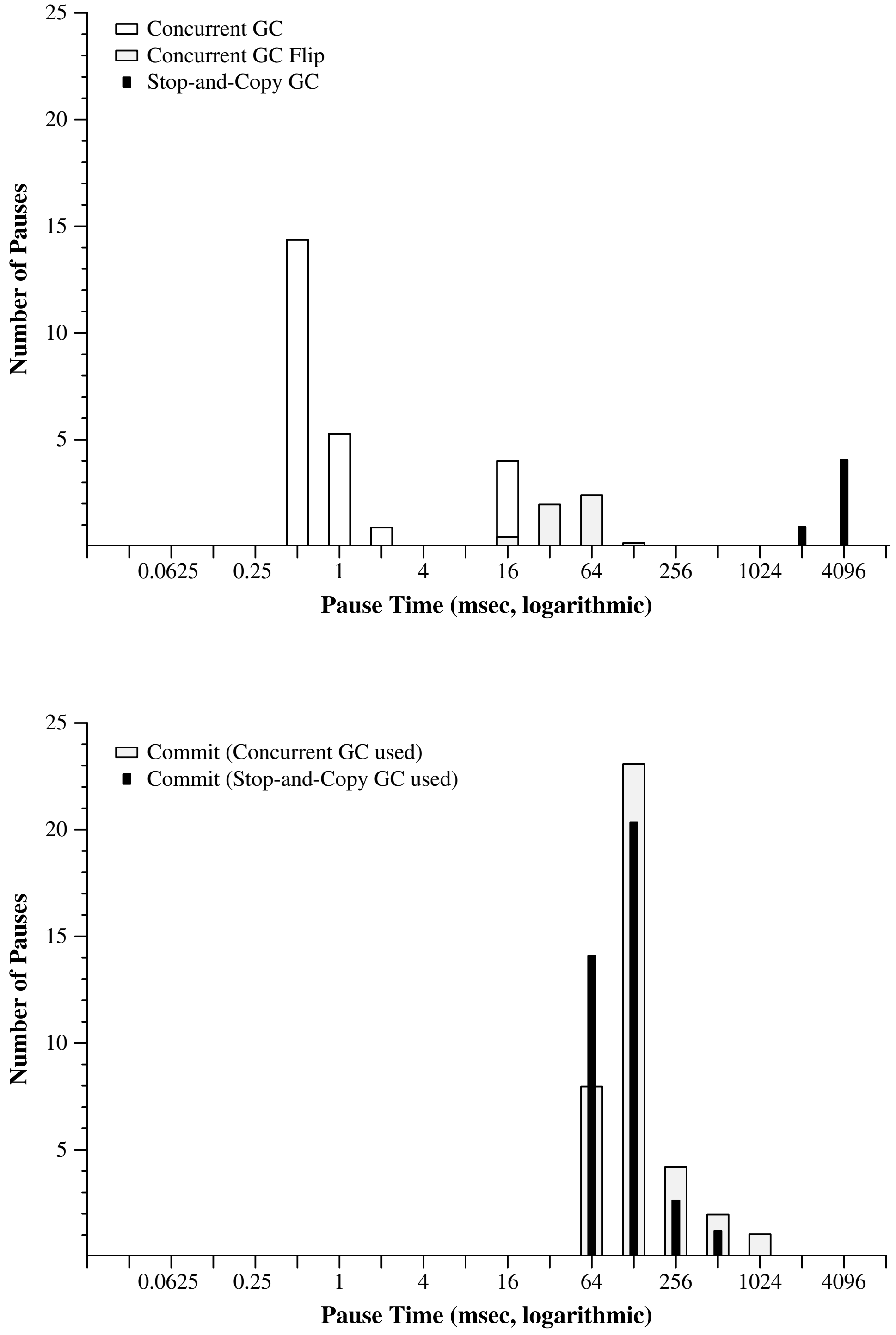


Figure 9.9: Compiler Pause Times (uniprocessor)

collection are shown in white and gray, and when running stop-and-copy collection in black.

As for the multiprocessor case, the first thing to note is that the pauses due to concurrent collection are more than an order of magnitude shorter than those due to stop-and-copy collection. Since the concurrent collector is often called without actually flipping, there are many more concurrent pauses, but most of them are quite short (i.e., less than 1 msec). Since the stop-and-copy collector is forced to flip the same number of times as the concurrent collector, the number of flip pauses, which are the most disruptive, is the same. Our earlier measurements for persistent GC throughput have already shown that the total time spent in concurrent collection is much less than for stop-and-copy collection. Looking at just the concurrent GC pauses, we see that the pauses that occur when flipping are longer than the non-flip pauses.

We also see that the concurrent GC pauses are about as long as the shortest `commit` pauses, but that the `commit` pauses are much more frequent. The `commit` pauses when concurrent collection is in use are skewed slightly to longer values compared to stop-and-copy; this is a result of the interference we noted in the section on GC throughput. This effect is considerably greater here than for the multiprocessor case seen earlier. This reflects the greater interference due to the need to share the CPU.

Finalization Pauses

To pinpoint the cause of the flip pauses, I measured the contribution of various parts of the persistent collector to the pause times. I found that the most important contribution comes from the part of the code that is executed only after the system has decided to flip. This code serves to “finalize” the flip, redirecting transitory heap pointers, logging the flip to RVM, and so on, as discussed in Subsection 5.5.3.

Figure 9.10 shows the contribution to the concurrent collector pauses of the finalization routine. It has essentially the same format as Figure 9.9, but includes only the pauses for the concurrent collector. Also, the scale on the y-axis is not the same.

The finalization routine pauses are almost as long as the pauses for the entire collector. That they are somewhat shorter is reflected in that the peaks at 32 and 64 msec are the same height in this case, but for the collector pauses the 64 msec peak is slightly larger. At this scale it is clear that on rare occasions, pauses longer than 128 msec occur. I observed that no other component of the pause time showed contributions longer than 32 msec, so it is also clear that any reduction in pause times should focus on the finalization code.

Volatile Pointer Redirection Pauses

To probe deeper, I measured the components of the finalization routine. They are: redirecting all the pointers in the volatile heap that point to the persistent from-space to point to the persistent to-space, redirecting the roots to point to the persistent to-space, recording the flip in RVM, and adjusting various internal data structures that must change on a flip. The details of these operations are found in Chapter 3. Again one component, redirecting the pointers in the volatile heap, dominates the observed pauses.

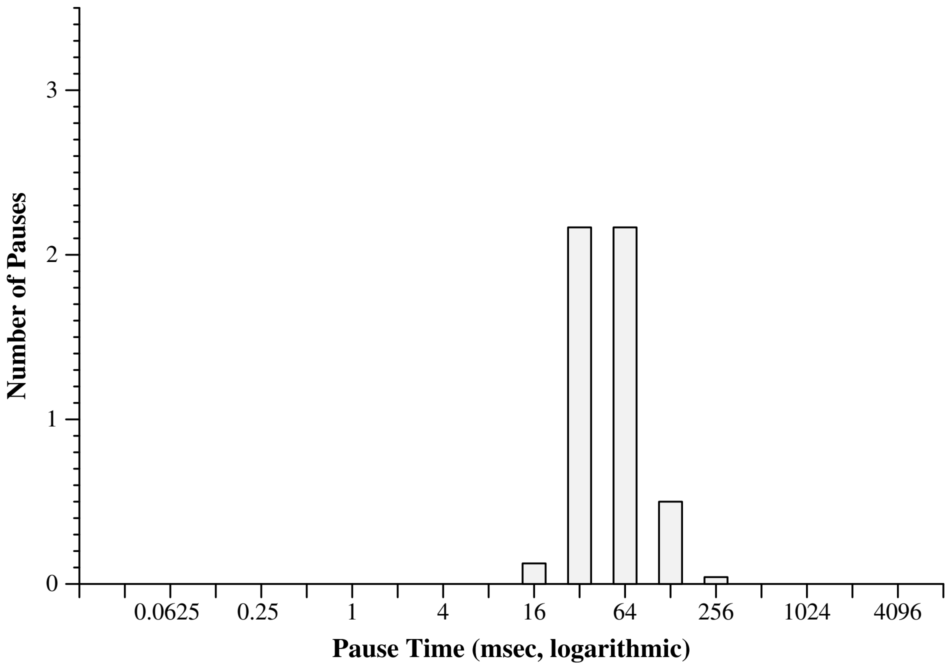


Figure 9.10: Finalization Routine Pause Times (uniprocessor)

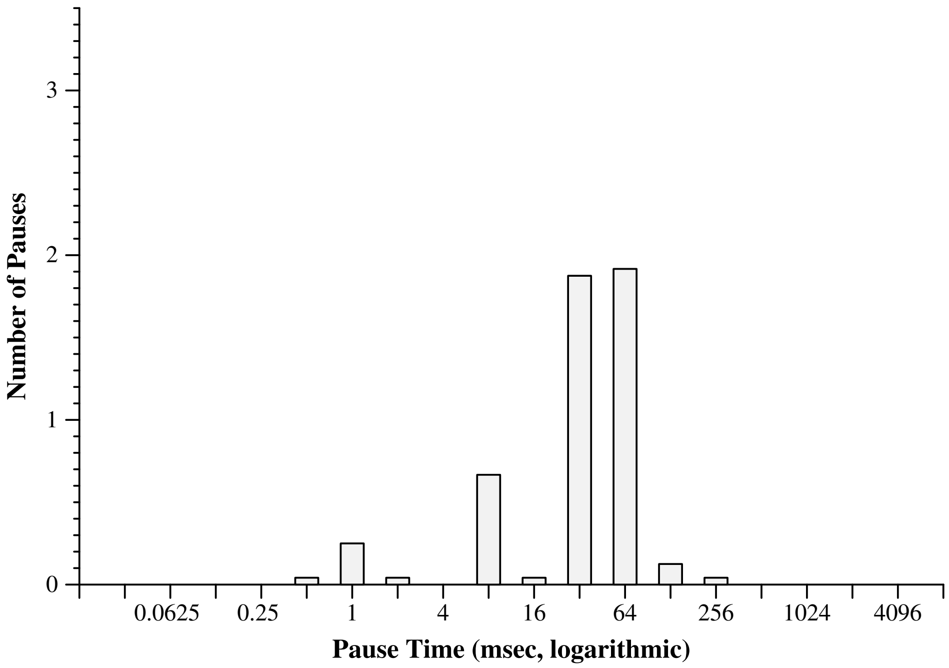


Figure 9.11: Volatile Pointer Redirection Pause Times (uniprocessor)

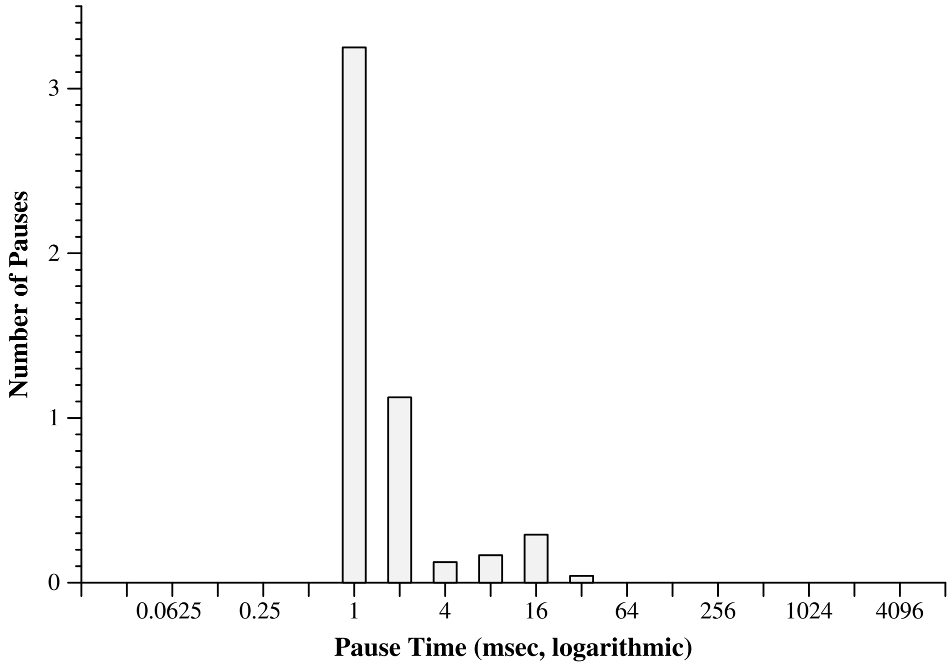


Figure 9.12: Copying Pause Times (uniprocessor)

Figure 9.11 shows the pauses due to redirecting volatile pointers in the same format as Figure 9.10.

The redirection pauses are almost as long as the finalization pauses. None of the other components of the finalization routine creates pauses of more than 16 msec. Fortunately, the routine for redirecting pointers can easily be optimized. Currently, it simply scans through all of the volatile to-space redirecting the relevant pointers. Unless there are a large number of such crossing pointers, a better implementation would be to maintain a table of these pointers so that only the pointers that needed to be updated would be manipulated and the entire to-space would not need to be examined. Another solution that should be explored is to coordinate flipping the persistent and volatile heaps. In this case, the to-spaces would be created with the correct pointers, and the flip would occur for both simultaneously. The issue of how to deal with crossing pointers is an important one, and I expect to address it more seriously in future work.

Copying Pauses

After the finalization routine, the next most important contribution to the GC pauses is from copying data from from-space to to-space. These pauses are short compared to those for finalization; presenting them reinforces the point that finalization is the first area that should be improved.

Figure 9.12 shows the contribution to the GC pauses from copying persistent data in the same format as Figure 9.10.

The pauses caused by copying persistent data are much shorter than those caused by redirecting the volatile pointers; only the longest copying pauses are longer than the shortest pauses due to redirection. Furthermore, the length of the copying pauses can be adjusted by changing the parameter that controls how much copying is done.

9.3 Summary

The goal of using the **comp** benchmark was to investigate, on a substantial program, all aspects of Sidney's performance, that is, GC and `commit` throughput and pause times. To achieve this goal, I adapted the SML/NJ compiler, a large widely used program, to store its state in the persistent heap. This conversion was made a great deal easier because of Sidney's support for orthogonal persistence, which allowed most of the state that needed to be persistent to be found implicitly. The benchmark itself used the modified compiler to compile files taken from the compiler itself. This resulted in a benchmark that had a small number (37) of long running transactions during which the collector was able to flip a small number of times (four or five).

The results of running **comp** provide strong support for the thesis. On both the multiprocessor and uniprocessor, concurrent collection results in modest speedups that are more than half of the maximum speedup possible. `Commit` is fast, less than 70 msec per `commit` on average. Also the cost of orthogonal persistence is small, 0.2% of the overall running time and only 7.5% of the cost of `commit`. Finally the pauses created by collection are short enough to be non-disruptive, less than 64 msec. Furthermore there are several possible ways to improve these times.

Chapter 10

The OO1 Benchmark

The **OO1** benchmark is a modified version of a standard object-oriented database (OODB) benchmark and serves several purposes in the dissertation. First, as an example of an OODB application, **OO1** helps us to evaluate Sidney as a building block for OODBs. Second, because **OO1** exercises all aspects of Sidney's design and implementation, it supplements the results of **comp**. Most importantly, **OO1** is used to study the behavior of Sidney as a function of the heap's livesize. Since the livesize is the most important factor influencing the cost of individual collections, studying its effect is essential for comparing concurrent collection to stop-and-copy collection and is important for understanding how Sidney will perform as persistent heaps become large.

OO1 is a synthetic benchmark that models the use of an OODB for a database of "parts" used to manufacture some product. **OO1** does not emphasize `commit` as heavily as **trans**, but it still spends most of its time committing or, as configured for testing, garbage collecting. Although the artificially heavy emphasis on `commit` and collection means that **OO1** will have higher runtime overheads than for real applications, it makes **OO1** a good stress test for Sidney.

The results of running **OO1** support the thesis. In Chapter 6, we saw that concurrent collection achieves substantial speedups compared to stop-and-copy collection. This chapter shows that the speedups come from overlapping I/O with computation, explaining why the speedups are not limited to the multiprocessor. In Chapter 6, we also saw that on the uniprocessor, the lengths of the concurrent collection pauses are not dependent on the livesize; furthermore, these pauses are much briefer than the pauses caused by stop-and-copy collection. In this chapter, we will see that the same results hold for the multiprocessor, and that the concurrent collector pauses on both platforms are also somewhat shorter than those caused by `commit`. However, the collector pauses are still longer than ideal. This chapter identifies the cause of the longer pauses as the need to scan the transitory heap when flipping the persistent heap. Possible solutions for this problem have already been discussed in the **comp** chapter, Section 9.2.2.

The chapter begins by discussing the **OO1** benchmark itself, along with some details, such as the GC parameters. The heart of the chapter are the detailed performance results. Measurements that relate to the throughput are presented first, followed by an examination of the latencies of collection and `commit`. The chapter ends with a brief summary.

10.1 Description of OO1

My goal in using **OO1** was two-fold. First, OODBs are an increasingly important part of database design and implementation. Since OODBs use much more flexible data types than more traditional databases, they need dynamic allocation of persistent data. Therefore, I wanted to assess how Sidney would perform in this area. I also wanted to have one benchmark that exercised all of Sidney's functionality and could also be used to study Sidney's dependence on the heap's livesize. Studying the dependence on livesize is important because the advantages of using concurrent collection are likely to increase as the amount of data that must be copied in a collection increases. The benchmarks **trans** and **coda** both exercise only limited aspects of Sidney, and **comp** has a livesize that changes during its execution. **OO1** avoids these issues and thus exercises all of Sidney. Additionally, systematic changes made to the livesize are not obscured by fluctuations inherent in the benchmark. These aspects of **OO1** make it an ideal benchmark for Sidney.

OO1 is a variant of the OO1 Engineering Database benchmark described by Cattell [13]. The original OO1 benchmark models an engineering application working on database of parts, performing traversals of the database, and adding parts. It is the first widely used OODB benchmark. As originally specified, OO1 never deleted any data from the database, making it a poor benchmark for a garbage collector. The version used here deletes parts so that on average the same number of parts are deleted as are added, keeping the livesize essentially constant. This change seems in the spirit of the original benchmark, and though it does preclude directly comparing the results here to those of others, the differences in programming language, transaction managers, and execution platforms would already make such a comparison impossible.

To adjust the livesize, I add varying amounts of data to the persistent heap such that the additional data are reachable from the persistent root. If no additional data are added, the livesize is just the amount needed to run **OO1**, about 4.5 MB. This livesize is 50% more than the average of 3.0 MB used by **coda** and 7.5 times the 600 KB used by **comp**. I add varying amounts of data up to a maximum total livesize of about 21 MB. The maximum was chosen to match the onset of paging for the uniprocessor. The reader may notice that this amount is much less than the total physical memory in the machine, 128 MB. The OS, other programs, and Sidney's text segment require 20 MB. To avoid paging, both the from-space and to-space must be backed by physical memory, accounting for two times the livesize, for a total of 42 MB. For technical reasons involving benchmark initialization, the transitory heap also needs to hold the live data as well as needing 2 MB for the active transitory data, for a total of 23 MB. The remaining 43 MB is used as buffer space by RVM.

In the measurements reported here, **OO1** is run for 50 iterations of the basic benchmark, resulting in running times of at least 90 sec. This time is long enough to allow the garbage collector to complete at least one collection of even the largest heaps. All of the numbers reported below are per iteration of **OO1**.

Subsection 6.2.1 of Chapter 6 discussed the issue of making the stop-and-copy collector track the points at which the concurrent collector flips. For **OO1**, a related issue arises in that for a given livesize the non-determinism of the concurrent collector results in a varying number of flips. For **coda**, I presented data for a variety of flip counts. If that approach were taken here, the need to also present the data as a function of livesize would make the data very hard to present and even harder to understand. Instead, I pick the most frequently occurring number of flips as the standard for each livesize, and I exclude data from runs that do not have this number of flips. As already seen in Subsection 6.1.1, omitting these runs leads to discontinuities in the data when the number of flips changes from one livesize to the next. It also makes data collection more difficult since many of the runs must be discarded. For the uniprocessor, the number of flips varies from two to one. For the multiprocessor, the number of flips varies from five to two.

One final issue is how the garbage collector parameters are configured. The less significant parameters are set as described in Section 6.2.1 of Chapter 6. The threshold for triggering both minor and major collections is set at 1 MB. Experimentation showed that larger values did not substantially speedup the program and picking somewhat modest sizes left more memory available to be used for varying the livesize. More important is the threshold for triggering persistent collection, which is set at 10 KB of allocation. A consequence of choosing such a small value is that the concurrent collector is essentially always running. Since the stop-and-copy collector is synchronized with the concurrent collector, it flips only when the concurrent collector does, not every 10 KB. Having the collector constantly running makes it easier to study the collector's behavior, but it also means that the collection overheads seen here will not be representative of ones that might be seen in practice. In practice, the overheads will be much lower.

10.2 Results of OO1

This section presents the results of the detailed performance analysis of **OO1**. It builds on the results already presented in Chapter 6, adding results for the multiprocessor, information about the distribution of values, and a detailed analysis of the factors influencing both the system's throughput and the pauses due to collection. The analysis covers four basic topics: the dependence of the throughput of **OO1** on the livesize on both the uniprocessor and multiprocessor, a detailed analysis of the throughput of the uniprocessor, the dependence of the collector and `commit` pause times on livesize on both the uniprocessor and multiprocessor, and a detailed analysis of the collector pause times on the uniprocessor. The detailed analysis parallel the presentation in Chapter 9 on **comp**. In summary, the results of this section are:

Dependence of Throughput on Livesize

> For both platforms, concurrent collection shows no systematic dependence on livesize and provides markedly better throughput than stop-and-copy collection. On the uniprocessor, the concurrent collector has a larger working set size, and thus at larger livesizes its throughput begins to degrade as the system begins to page. On both platforms, stop-and-copy collection shows a linearly increasing dependence on livesize, as well as a strong dependence on the number of flips achieved.

Details of Throughput

The detailed throughput results are reported for the uniprocessor, with a livesize of 4.5 MB. The overall speedup due to concurrent collection is 9%. Concurrent collection results in essentially all of the collection-related overheads being transferred to the GC and I/O threads. However, only about a third of the maximum possible speedup is observed. Some of the slowdown can be traced to greater CPU use by the GC and I/O threads, and some to an increase in CPU use by parts of the runtime other than collection. The reasons for the latter increases are unclear.

Dependence of Latencies on Livesize

For both platforms, the lengths of the pauses due to concurrent collection show no systematic dependence on the livesize and are at least an order of magnitude shorter than those caused by stop-and-copy collection. The pauses due to concurrent collection are also shorter than those due to `commit`. The length of the pauses due to stop-and-copy collection show a linearly increasing dependence on livesize.

Details of Latencies

The detailed latency results are presented for the uniprocessor with a 4.5 MB livesize. Except for the length of the pauses being examined, they show a striking similarity to the same results for **comp**. The longest pauses are longer than one second and occur when the persistent heap is flipped. The source of these long pauses lies in the part of the flip finalization routine that scans the transitory heap to redirect pointers in the persistent from-space to the persistent to-space. Several solutions for this problem were proposed in Chapter 9.

10.2.1 Dependence of Throughput on Livesize

One of the most important results for **OO1** is how its elapsed time depends on livesize. For the uniprocessor, these results were presented in Chapter 6. Here, I present the results for similar experiments on the multiprocessor, including the quartiles as "error bars." I also repeat the uniprocessor results with the addition of the quartiles.

Figure 10.1 shows the dependence of the elapsed time of **OO1** on the livesize for the multiprocessor. The y-axis shows the number of seconds taken to complete a single iteration of the benchmark. The x-axis shows the livesize in megabytes. The median values when the stop-and-copy collector is used are shown as boxes and when the concurrent collector is used as triangles. The "error bars" show the quartiles.

The basic trends are the same as already presented for the uniprocessor. Concurrent collection shows no systematic dependence on the livesize and also offers uniformly better performance than stop-and-copy collection. Even in the worst case, concurrent collection is about a 30% improvement over stop-and-copy collection. For stop-and-copy collection, as in Chapter 6, we see the sawtoothed shape that results from changing the number of flips. For the multiprocessor, we go from five flips at a livesize of 4.5 MB to two flips at a livesize of 21 MB. For a particular number of flips, the stop-and-copy collector shows the

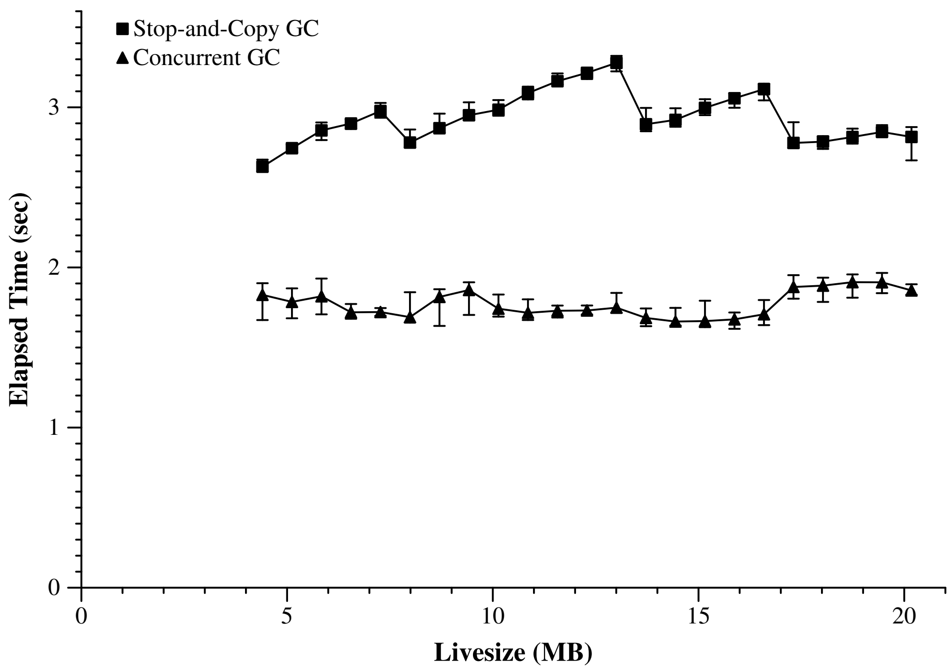


Figure 10.1: **OO1** Times versus Livesize (multiprocessor)

expected linear dependence on livesize. The quartile values are tightly clustered around the medians, although the concurrent collector shows somewhat greater variation. This variation probably reflects the nondeterministic nature of the concurrent collector.

Figure 10.2 shows the basic throughput of **OO1** on the uniprocessor in the same format as Figure 10.1. Except for the addition of the quartiles, this figure duplicates the one in Section 6.1.1.

This measurement was discussed in Section 6.1.1. To recapitulate, first note that the throughput of the stop-and-copy collector has a strong dependence on the livesize while the concurrent collector does not. Furthermore, concurrent collection provides substantially better throughput than stop-and-copy collection. The increase in elapsed time for the concurrent collector beginning at 18 MB is due to the onset of paging. The addition of the quartiles shows that the distributions are quite tight around the median, differing from the median by only a few percent. It also shows that some of the minor irregularities seen in the plots are probably a result of small errors in the median, rather than anything more systematic.

10.2.2 Details of Throughput

This subsection examines the details of the overheads that contribute to the throughput of **OO1**. Rather than attempting to measure detailed quantities for all the livesize parameters, I looked at two livesize values, the minimum of 4.5 MB and close to the maximum at 19 MB. These experiments were conducted on both the uniprocessor and the multiprocessor. All of the results are similar and support the same general conclusions about Sidney's behavior.

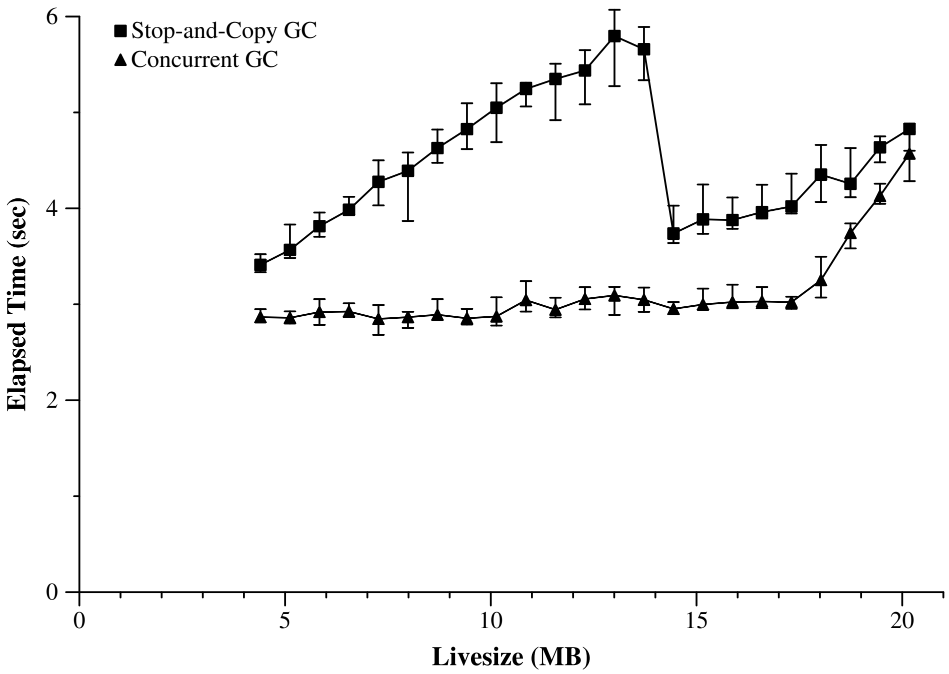


Figure 10.2: **OO1** Times versus Livesize (uniprocessor)

In the interest of brevity, I have included only the uniprocessor results for the 4.5 MB case here. This subsection parallels the discussion of **comp** in Chapter 9, but omits the detailed examination of `commit`.

Total Time and Persistent GC

The most important throughput-related measurement is the total time for both the stop-and-copy and concurrent collector cases. This measurement both quantifies the overall speedup and gives an indication as to the source of the speedup. The other result presented here is the time spent in persistent GC. The difference in collection times between the stop-and-copy and concurrent collectors provides a bound on the speedup possible and gives an idea about how successful Sidney is in off-loading work from the client thread to the GC and I/O threads.

Table 10.3 contains the results for both total time and persistent garbage collection in the standard format.

When concurrent collection is used, the total time has a speedup of 0.30 sec elapsed (-0.08 sec CPU), which is a speedup of 9% relative to stop-and-copy collection. Since the processor is a uniprocessor, all of the elapsed time improvement is due to I/O overlap. The time spent by the garbage collector running in the client thread (mostly when flipping) is negligible, only 0.02 sec per iteration. On the other hand, the time spent in the stop-and-copy collector is substantial, 1.41 sec elapsed (0.39 sec CPU), which is 43% of the total elapsed time. Most of this time, 1.02 sec, is spent doing I/O. Overlapping this I/O with computation is the primary throughput-related benefit of using Sidney, even on the

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Total Time		Concurrent	2.91	100.0	-2.7	0.7
Total Time	(CPU)	Concurrent	1.63	100.0	-3.9	0.6
Total Time		Stop&Copy	3.21	100.0	-2.6	1.9
Total Time	(CPU)	Stop&Copy	1.55	100.0	-2.0	1.5
Persistent GC		Concurrent	0.02	0.8	-6.0	9.4
Persistent GC	(CPU)	Concurrent	0.02	1.5	-10.0	5.7
Persistent GC		Stop&Copy	1.41	43.0	-4.9	6.5
Persistent GC	(CPU)	Stop&Copy	0.39	25.0	-3.1	9.2

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Total Time	-0.30	0.08	1.30	1.70
Persistent GC	-1.39	-0.37	0.00	1.02

Table 10.3: **OO1** Total and Persistent GC Times

multiprocessor. The total speedup is much less than the speedup seen for collection, which means that the other parts of the system experience a slowdown when concurrent collection is introduced. The slowdown of the total time relative to the gains in persistent collection is 1.09 sec elapsed; at least 0.37 sec of this amount comes from the CPU time transferred from the client thread when using stop-and-copy collection, to the GC and I/O threads when using concurrent collection.

Runtime Overheads

The total speedup measured falls a good deal short of that offered by the change in collection times. The next set of measurements attempts to gain some insight into why the full speedup is not achieved by looking at the costs of various parts of the client thread. At the highest level is the time spent in the runtime and, by comparison with the total time, the time in user code. Additional measurements establish the times spent in `commit`, major collection, and minor collection. These, together with the time spent in persistent collection, comprise the time spent in the runtime.

Table 10.4 shows the results of measuring the runtime and its components. The measurements are presented in decreasing order of elapsed time. The rows labeled *Runtime* are for the entire runtime, those labeled *Commit* are for `commit`, those labeled *Minor Collection* are for the minor collections, and those labeled *Major Collection* are for the major collections. The measurements for persistent collection are not repeated.

The runtime system shows a speedup of 0.44 sec elapsed (0.00 sec CPU) which represents a slowdown relative to the persistent collector of 0.95 sec elapsed (0.37 sec CPU). This slowdown is 87% of the total elapsed time slowdown. In the stop-and-copy collector

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
Runtime		Concurrent	2.55	88.0	-1.4	1.3
Runtime	(CPU)	Concurrent	1.33	82.0	-3.6	0.6
Runtime		Stop&Copy	2.98	93.0	-4.6	3.9
Runtime	(CPU)	Stop&Copy	1.33	86.0	-3.1	2.0
Commit		Concurrent	2.23	77.0	-2.2	0.8
Commit	(CPU)	Concurrent	1.17	72.0	-4.5	1.3
Commit		Stop&Copy	1.50	47.0	-0.6	0.4
Commit	(CPU)	Stop&Copy	0.84	54.0	-0.6	0.3
Minor GC		Concurrent	0.23	8.1	-4.1	2.9
Minor GC	(CPU)	Concurrent	0.20	12.0	-1.4	0.3
Minor GC		Stop&Copy	0.15	4.5	-0.6	0.4
Minor GC	(CPU)	Stop&Copy	0.15	9.4	-1.2	0.4
Major GC		Concurrent	0.07	2.5	-6.5	3.7
Major GC	(CPU)	Concurrent	0.06	3.9	-9.9	4.5
Major GC		Stop&Copy	0.05	1.5	-0.5	0.4
Major GC	(CPU)	Stop&Copy	0.05	3.1	-0.6	0.6

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
Runtime	-0.44	0.00	1.20	1.70
Commit	0.72	0.33	1.10	0.66
Minor GC	0.09	0.05	0.04	0.00
Major GC	0.02	0.01	0.00	0.00

Table 10.4: Runtime Overheads

case, the times for the user code can be found by subtracting the runtime costs from the total time. In the concurrent collector case, because the elapsed time will also be affected by the GC and I/O threads' use of the CPU, the elapsed time for the user code cannot be found this way. The CPU clock is maintained on a per thread basis, so we can find the user code CPU time this way. User code takes 0.23 sec elapsed (0.22 sec CPU) when using stop-and-copy collection, and 0.30 sec CPU when using concurrent collection. Thus, user code shows a slowdown of 0.08 sec CPU or 36% when concurrent collection is used. For the stop-and-copy case, the user code is 8% of the elapsed time and 17% of the CPU time, showing that **OO1** is in fact very `commit`- and GC-intensive, more so than is likely to be true for a real application.

Now consider the detailed components of the runtime. For the concurrent case, `commit` is the dominant component taking 2.23 sec elapsed (1.17 sec CPU) and accounting for 77% of the total elapsed time. For the stop-and-copy case, persistent collection is almost as expensive as `commit`, but `commit` takes 1.50 sec elapsed (0.84 sec CPU) and accounts for almost 50% of the overall elapsed time. Comparing stop-and-copy to concurrent, `commit` shows a slowdown of 0.72 sec elapsed (0.33 sec CPU). The elapsed time slowdown could come both from sharing the CPU with the other threads and from I/O interference as well as the increase in CPU time. As was the case for **comp**, the CPU slowdown is puzzling. This slowdown is 66% of the total elapsed time slowdown and 73% of the total CPU slowdown (for the client thread), which is close to the relative contribution of `commit` to the concurrent times. Major and minor collection are much less significant contributors to the runtime overheads and to the slowdowns. The slowdown for minor collection is 0.09 sec elapsed (0.05 sec CPU), which is 8% of the total elapsed time slowdown and 11% of the total CPU slowdown. Thus, minor collection shows slightly more slowdown relative to its contributions to the stop-and-copy results than `commit`. Major collection shows a slowdown of 0.02 sec elapsed (0.005 sec CPU), or about 2% relative to the totals, roughly in line with its stop-and-copy contributions. Taken together, the slowdowns for `commit` and major and minor collection are 0.83 sec elapsed (0.39 sec CPU), as compared to 0.44 sec elapsed (0.0 sec CPU) observed for the runtime overall.

Persistent GC Overheads

Now we look at the GC and I/O threads. Essentially the only overhead in the GC thread is copying the live data, a cost that is present in all copying collectors. Since these overheads involve using the CPU, we would expect that transferring them to the GC thread would increase the overall elapsed time, due to overheads caused by incremental processing and context switching overheads. The overheads in the I/O thread are all due to persistence: writing the heap to disk, and truncating the RVM log. Since these overheads involve a great deal of I/O, transferring these costs to the I/O thread is the source of the speedup from using concurrent collection. For both threads I compare the costs of the concurrent collector case to the equivalent parts of the stop-and-copy case.

Table 10.5 shows the results of measuring the costs of copying and writing to-space. The rows marked *GC Copying* are for copying and scanning the data. The rows marked *To-Space I/O* are for all of the I/O related activities involved in writing to-space, those marked *Truncate* are just the cost of truncating the RVM log, and those marked *Write Heap* are for writing the heap.

For concurrent collection, the cost of copying is 0.29 sec elapsed (0.26 sec CPU). Since the thread does no I/O, the difference between the elapsed and CPU times, 0.03 sec, is due to sharing the CPU. The CPU time for the concurrent case is more than a factor of four times that of the stop-and-copy case.

The concurrent collector takes 1.66 sec elapsed (0.39 sec CPU) more than the stop-and-copy collector to write to-space, which is more than a factor of two. In fact, the I/O thread is almost always busy doing some kind of work. Of that time, twice as much is used in truncating the RVM log than on writing the heap, strongly suggesting that it would be worthwhile to eliminate the need to truncate the RVM log.

Quantity Measured		Collector Used	Median (sec)	Total (%)	25th (%)	75th (%)
GC Copying		Concurrent	0.29	10.0	-1.8	2.2
GC Copying	(CPU)	Concurrent	0.26	16.0	-0.9	1.1
GC Copying		Stop&Copy	0.06	2.0	-2.3	1.7
GC Copying	(CPU)	Stop&Copy	0.06	4.2	-2.0	2.0
To-Space I/O		Concurrent	2.88	99.0	-3.0	2.2
To-Space I/O	CPU	Concurrent	0.66	40.5	-7.2	5.9
To-Space I/O		Stop&Copy	1.22	39.0	-4.1	11.0
To-Space I/O	(CPU)	Stop&Copy	0.27	17.0	-5.8	13.0
Truncate		Concurrent	1.92	66.0	-2.7	1.4
Truncate	(CPU)	Concurrent	0.59	36.0	-1.8	1.3
Truncate		Stop&Copy	0.92	28.0	-15.0	6.5
Truncate	(CPU)	Stop&Copy	0.26	17.0	-18.0	5.8
Write Heap		Concurrent	0.94	33.0	-1.9	4.4
Write Heap	(CPU)	Concurrent	0.06	3.5	-8.9	5.0
Write Heap		Stop&Copy	0.40	12.0	-2.2	1.3
Write Heap	(CPU)	Stop&Copy	0.03	2.2	-2.7	3.4

Delta	Elapsed (sec)	CPU (sec)	Concurrent (sec)	Stop&Copy (sec)
GC Copying	0.23	0.20	0.03	0.00
To-Space I/O	1.66	0.39	2.22	0.95
Truncate	1.00	0.33	1.33	0.66
Write Heap	0.54	0.02	0.89	0.37

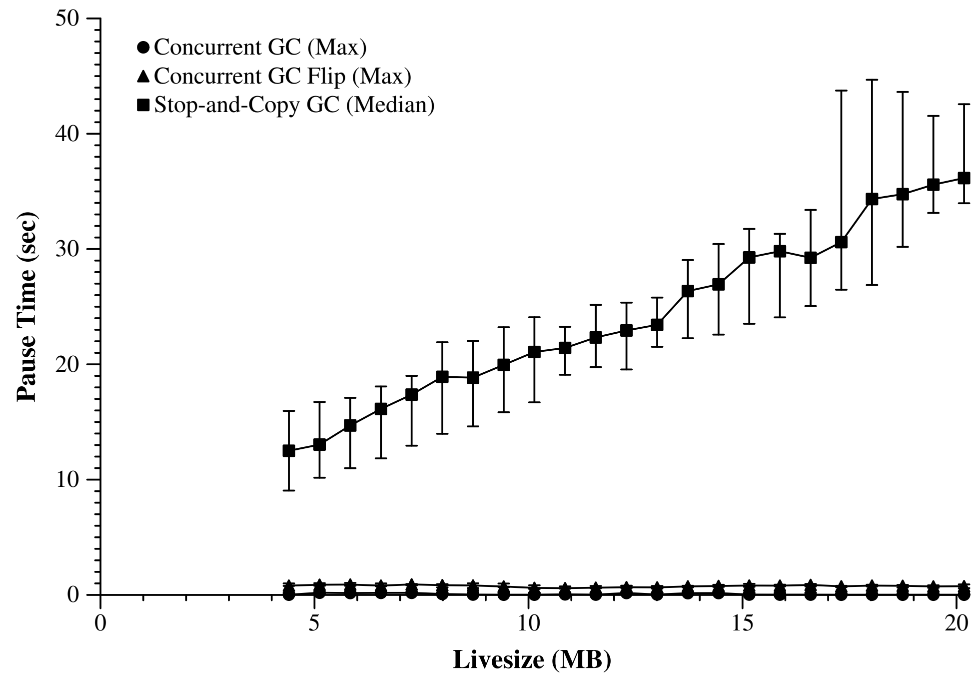
Table 10.5: GC Overheads

10.2.3 Dependence of Latencies on Livesize

It is a bonus that concurrent collection results in better throughput for **OO1**; the principal reason for introducing the technique is to eliminate the disruptive pauses commonly associated with collection. In this subsection, I examine the dependence of the pause times for GC and `commit` on the livesize. Chapter 6 contains these results for collection pauses on the uniprocessor. This subsection omits those results, presenting instead the multiprocessor collection pauses as well as the `commit` pauses for both machines. These results include the quartiles, unlike those in Chapter 6.

GC pauses

Figure 10.6 shows the length of the pauses that result from collection as a function of livesize for the multiprocessor. The upper plot gives the results for stop-and-copy collection as well



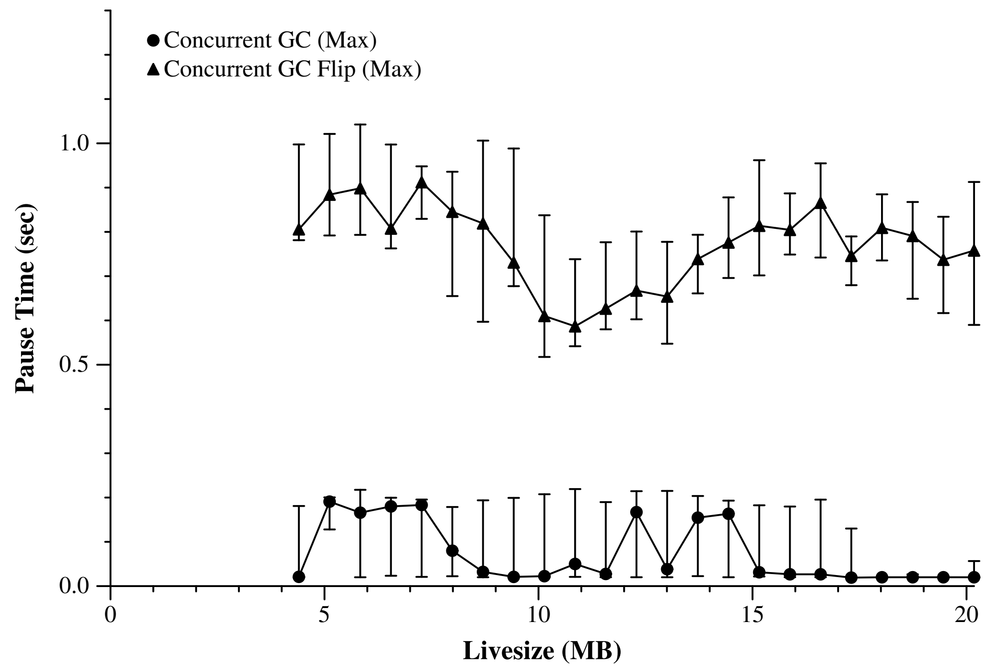


Figure 10.6: **OO1** Persistent GC Pause Times versus Livesize (multiprocessor)

as for the concurrent collector. The lower plot omits the stop-and-copy data, allowing the scale of the y-axis to be greatly expanded. The y-axis is the length of the pause in seconds. The x-axis specifies the livesize in megabytes. The value of the quartiles are shown by the "error" bars. The quantities marked *Median* are found by taking all the pauses of the given type from all runs and then computing the median and quartiles. The quantities marked *Max* are computed by first finding the maximum pause of the given type for each run and then computing the median and quartiles of these maximums across all runs. Section 6.2.3 discusses the reasons for using this method.

The top plot shows that the stop-and-copy pauses are much longer than the concurrent pauses and show a linear dependence on the livesize as expected. The sawtooth seen for the throughput measurement is not observed here because the pause times do not depend on the number of flips, but only on the livesize. The lower plot shows that the flip pauses are much longer than the non-flip pauses, but that neither type of pause shows a systematic dependence on the livesize. There is no reason to believe that the shape of the curves in the bottom plot has any particular significance. Although they are at least an order of magnitude shorter than the stop-and-copy pauses, the flip pauses for the concurrent collector are still well over half a second, long enough to be disruptive to a human user. The reason for these long pauses is investigated in the next subsection.

Addition of `Commit` Pauses

The next measurements look at the pauses created by `commit` and compares them to those created by collection. Although the concurrent collection pauses are longer than would be preferred, they are much less frequent than the `commit` pauses, occurring at most 5 times as compared to 50 times for `commit`. Thus, if they are of the same general length as the `commit` pauses, they will be less disruptive. In practice, with less aggressive collection parameters, the collection pauses would be even less disruptive since they would be less frequent.

Figure 10.7 shows the `commit` and collection pause times for the uniprocessor and Figure 10.8 for the multiprocessor. These plots are similar to Figure 10.6, but they include the median pause values for `commit` when the system is using both kinds of collection, and they omit the non-flip pauses for concurrent collection. For the concurrent flip pauses, the medians are reported rather than the maximums. This change avoids repeating results and allows the medians to be compared to the maximums previously presented.

The upper plots show that the stop-and-copy pauses are much longer than the `commit` pauses, more than factor of 20 for the uniprocessor and a factor of 10 for the multiprocessor. At the scale of the upper plots, there is no noticeable difference between the length of `commit` pauses in the systems using stop-and-copy collection or concurrent collection. The lower plots show that the flip pauses are somewhat shorter than the `commit` pauses. The `commit` pauses for the stop-and-copy based system are shorter than the `commit` pauses when concurrent collection is used. We saw in Section 10.2.2 that using concurrent collection results in a significant increase in the time for `commit`, so we would expect the pauses to be longer. Notice that the distribution of values is much broader for commits with concurrent collection, indicating that the interference of the collector makes the times for `commit` more

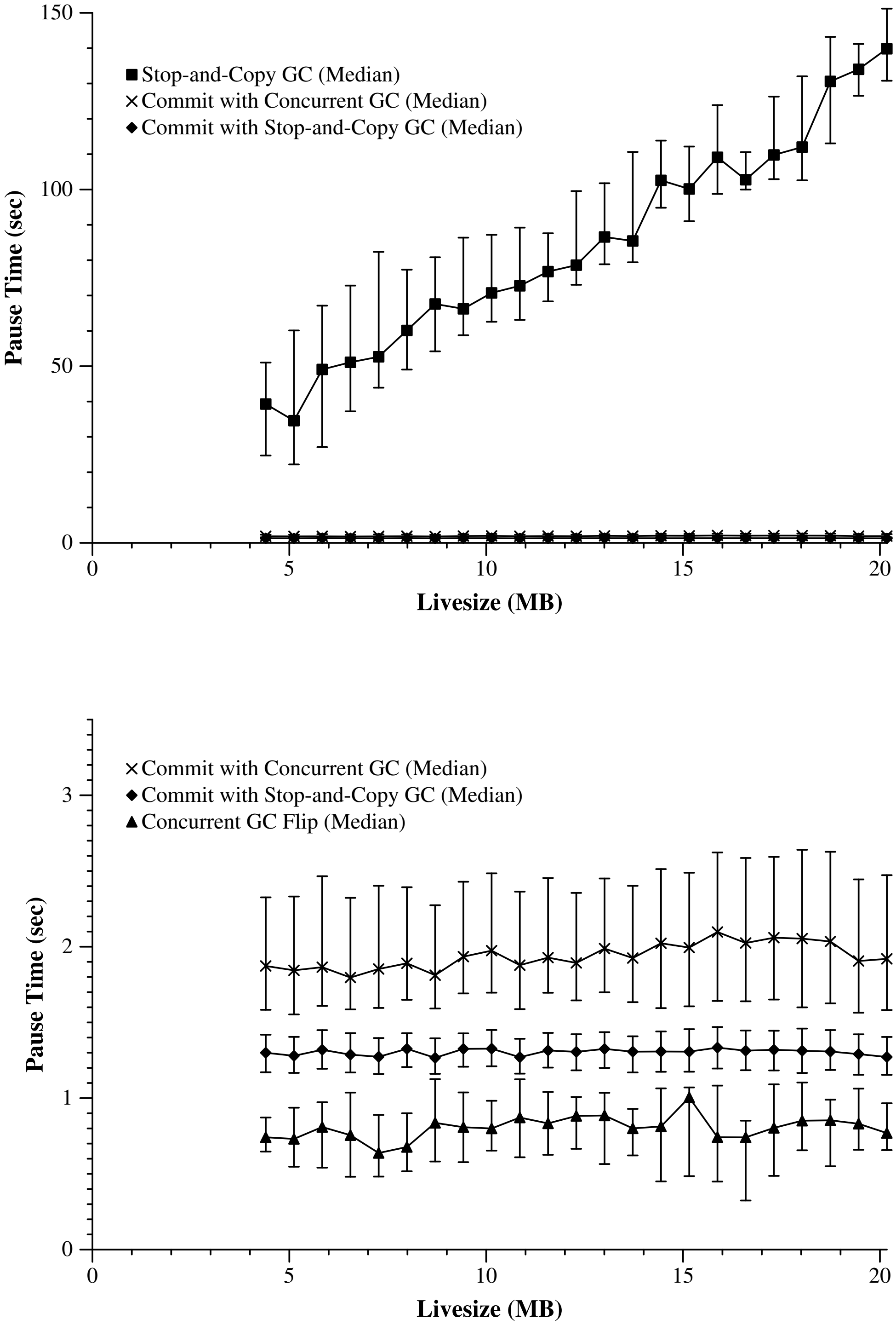
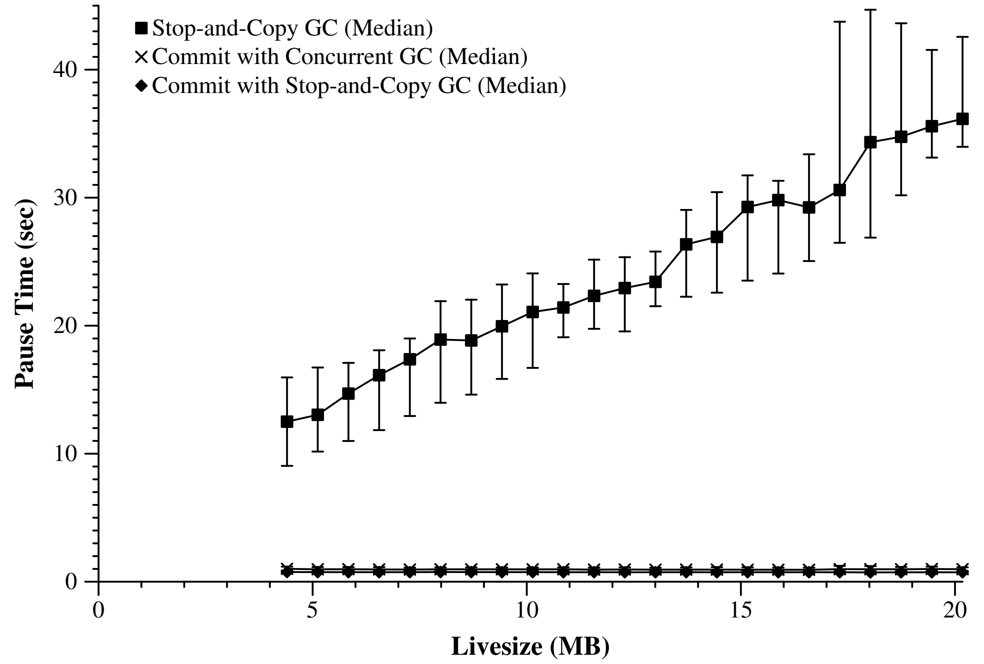


Figure 10.7: **OO1** Commit Pause Times versus Livesize (uniprocessor)



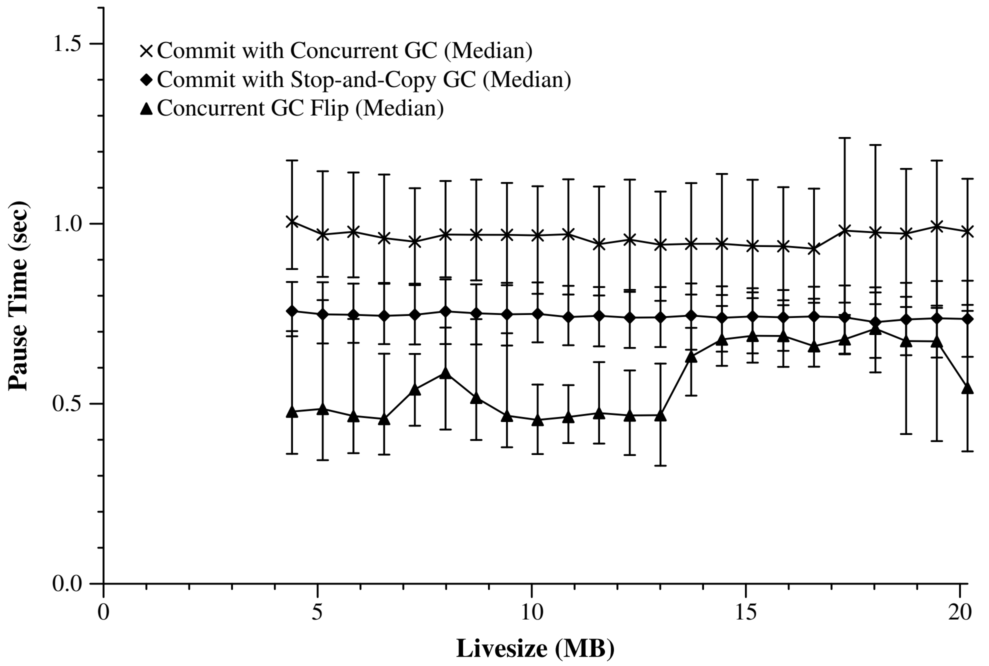


Figure 10.8: **OO1** GC and Commit Pauses versus Livesize(multiprocessor)

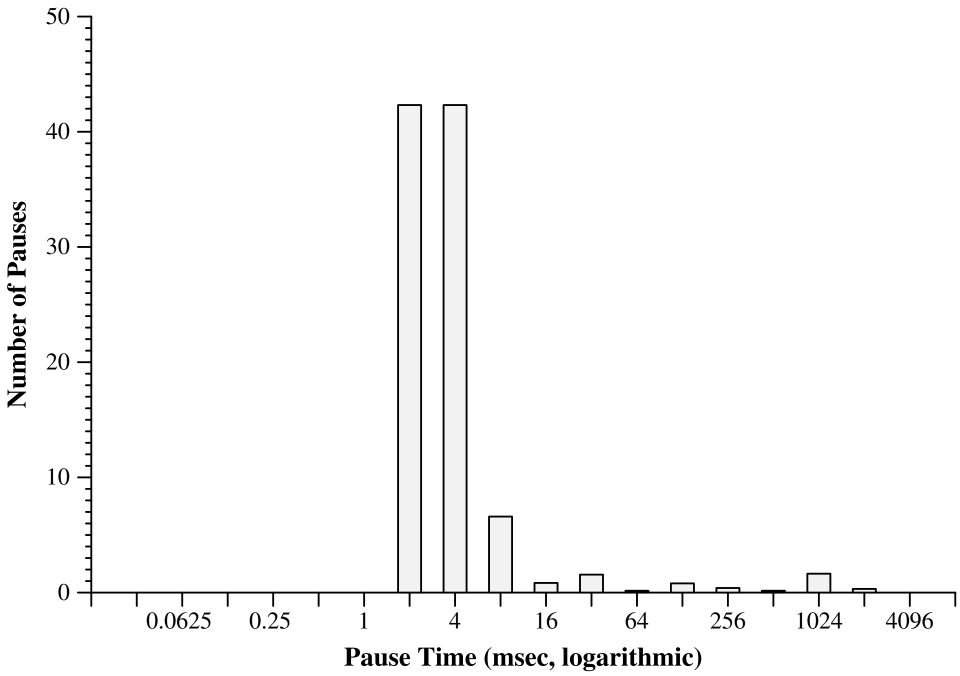


Figure 10.9: Concurrent GC Pauses

variable. I don't have a explanation for the apparently systematic increase in the flip pause time for the multiprocessor beginning at around 15 MB. However, there is not a similar increase in the maximum pause times, as seen in Figure 10.6.

10.2.4 Details of Latencies

Although they are of bounded length, shorter than the `commit` pauses, and much shorter than stop-and-copy pauses, the concurrent collection pauses are still longer than preferred. To understand why the pauses are as long as they are, I followed essentially the same process as for **comp** in Chapter 9. First, I examined all of the pauses and determined that the flip pauses were the source of the longest ones. Next, I examined the components of the flip pause and found that the finalization routine is the dominant component of the flip pauses. Finally, I examined the components of the finalization routine and found the dominant factor is scanning the volatile heap for pointers into persistent to-space.

In addition to presenting the pause times just mentioned, I also present the pauses caused by copying data in the client thread. They are the longest pauses that are not a part of finalization, and serve to show that only the finalization-related pauses are important to consider at this time, as well as to explain the many very short pauses. All of the results presented here are for the uniprocessor; the multiprocessor results are similar. The minimum livesize, 4.5 MB, is used to maximize the number of flips, but the results hold in general.

Figure 10.9 shows the distribution of all GC pauses. The y-axis is the number of pauses of a given duration. The x-axis is logarithmic in the pause duration in milliseconds.

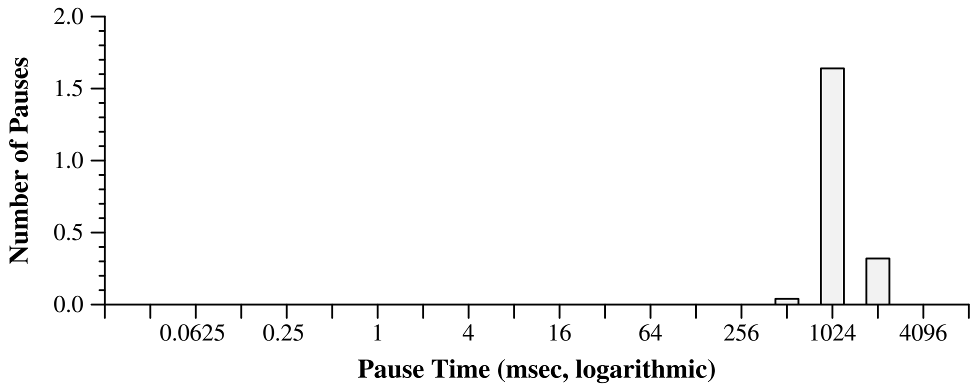


Figure 10.10: Concurrent GC Flip Pauses

The pauses have been placed into logarithmic bins (see Section 6.1.2). This plot shows a form that by now should be familiar: There are many short pauses caused by the need to synchronize briefly the client and GC threads, and there is a small number of much longer pauses that result when the collector flips.

Figure 10.10 shows the collection pauses that result from the collector flipping. This plot has the same form as Figure 10.9 but with a different scale for the y-axis. Comparing this plot to Figure 10.9 shows that the longest pauses are a result of flipping, as expected. The plot also shows that there were pauses for two flips, which is the correct number for this livesize. Also, remember these results come from averaging the results of at least 24 runs, which accounts for non-integral nature of the numbers of pauses.

Figure 10.11 shows, in the same format and scale as Figure 10.10, the part of the flip pause caused by the finalization routine. Comparing this plot to Figure 10.10 shows that finalization accounts for the flip pauses quite well. No other component of collection has contributions to the pause times of even as much as 128 msec.

The final level of detail leads to identifying the primary cause of the finalization pauses as the need to scan the transitory heap during a flip. Figure 10.12 shows the contribution to the pauses from this factor in the same format as Figure 10.10. Comparing this plot to Figure 10.11 shows that this factor is the main contributor to the finalization pauses, and

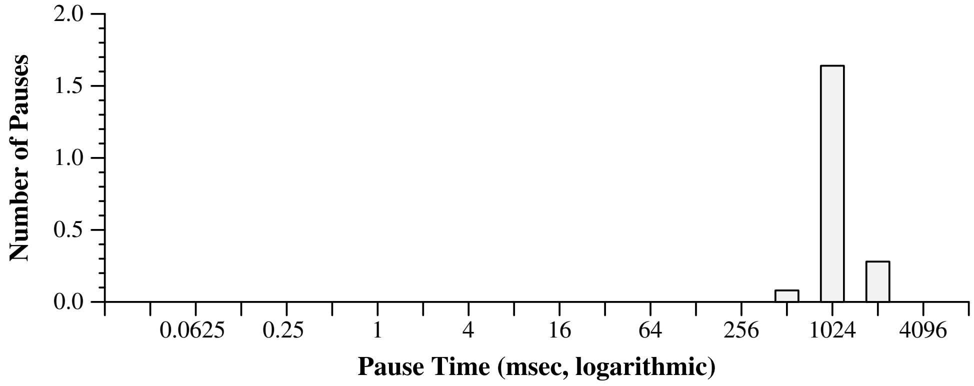


Figure 10.11: Finalization Pauses

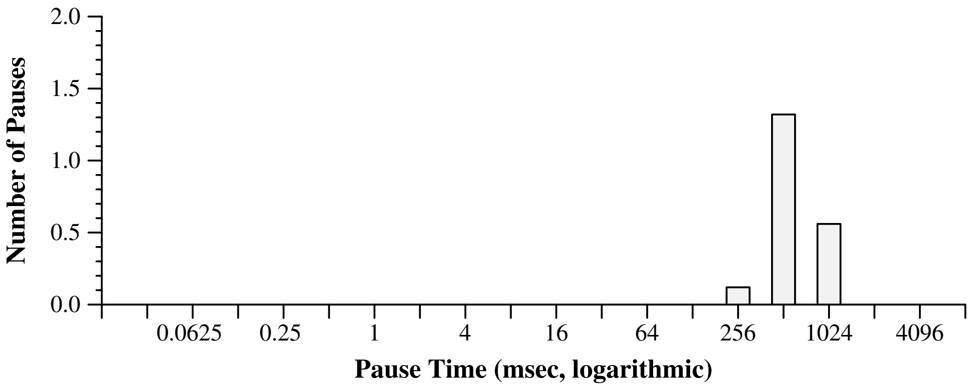


Figure 10.12: Transitory Heap Scan Pauses

thus to the long flip pauses. Other components of the finalization routine show lengths less than 256 msec. The shifts in the position of the peaks indicates that some contribution from other factors is present.

Figure 10.13 shows part of the collection pauses associated with copying and scanning data in the same format but different scale as in Figure 10.10. From this plot we see the main source of the many small pauses seen in Figure 10.9. It is also clear that these pauses are not an important factor in determining the maximum pause time.

Clearly, focusing on the time to scan the transitory heap is the way to reduce the pause times. Strategies for reducing this time have already been discussed in Chapter 9.

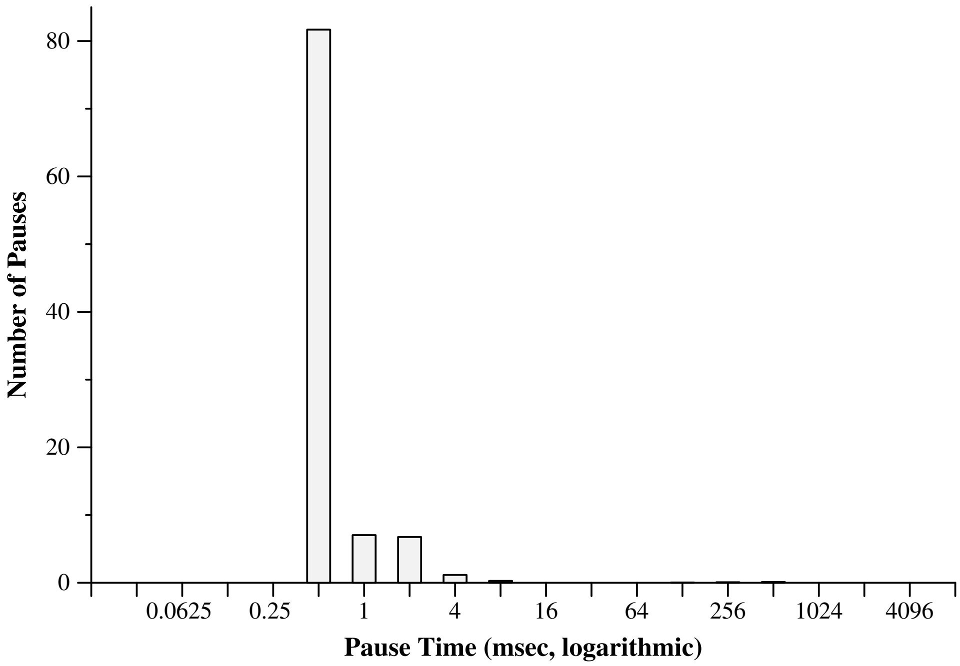


Figure 10.13: GC Copying Pauses

10.3 Summary

OO1 is a popular OODB benchmark that mimics an engineering application using a database of “parts,” traversing the data, and adding parts. **OO1** is derived from OO1 by having parts be deleted as well as added, thus making it suitable for benchmarking a garbage collector. **OO1** serves several purposes. First, it tests Sidney in one of its possible domains of application, OODBs. Second, because it uses all of Sidney's capabilities, **OO1** provides another full-range benchmark to complement **comp**. Finally, because the number of parts deleted balances the number added, it has a constant livesize. This allows the dependence of Sidney on livesize to be studied easily by just adding “extra” live data unrelated to the benchmark. Studying the dependence on livesize is important because stop-and-copy collection becomes more disruptive as livesize increases.

The results of running **OO1** are favorable to Sidney. The elapsed times for concurrent collection are not dependent on livesize and are substantially lower than those for stop-and-copy collection. The elapsed times for stop-and-copy collection show a linear increase with livesize and a strong dependence on the number of flips as well. Detailed examination of **OO1** throughput shows that the speedup for concurrent collection is the result of overlapping I/O with computation, although only about half of the maximum possible speedup is achieved. The pause times for concurrent collection also show no dependence on livesize and are at least an order of magnitude better than for stop-and-copy collection. They are also shorter and much less frequent than the pauses due to `commit`. The stop-and-copy pauses show the expected linear increase with livesize. Unfortunately, the concurrent collection pauses are still longer than half a second and thus likely to be disruptive. A careful examination of the source of the longest pauses shows that as in the case of **comp**, the longest pauses in running **OO1** are caused by the need to redirect pointers in the transitory heap into persistent from-space to persistent to-space. The **comp** chapter discusses possible solutions to this problem.

Chapter 11

Related Work

This chapter covers related work. In the design chapters, I offered a comparison of Sidney's design and architecture to the primary design alternatives, deferring some detailed discussion to this chapter. In addition to those points, here I also consider related work that does not directly bear on detailed design decisions.

The chapter begins with a discussion of work relating to transactions and general-purpose persistence. It then considers work relating to each of the key implicit mechanisms, touching on both implementation and performance evaluation related issues.

11.1 Features

In this section, I discuss work related to both of the key features of Sidney, general-purpose persistence and transactions. Both of these topics (and especially transactions) have an extensive literature, which I do not attempt to survey. Instead I focus on the issues most directly relating to Sidney.

11.1.1 General-Purpose Persistence

General-purpose persistence systems can be broadly classified into ones that support implicit techniques, orthogonal persistence in particular, and ones that use explicit techniques. I have considered one explicit system, based on RVM and RDS, in the performance evaluation and discussed them in general in the design chapters. Another good example of such a system is the Exodus storage manager and the E programming language [11, 47]. Since Sidney is an implicit system, I do not consider explicit systems further.

The pioneering work on defining and supporting orthogonal persistence is that of PS-Algol [5] and its follow-up language, Napier88 [38]. These languages support essentially the same model as Sidney, with writes logged automatically, a garbage collected store, and persistence that is orthogonal to other properties of data and determined by reachability from a persistent root. These languages have inspired many other systems that share all or some of these features. Sidney is not unique in the interface it provides, only in the performance it achieves.

Large Persistent Heaps

One issue that has received considerable attention in the persistent programming language community has been providing support for large heaps [5, 39]. In the design of Sidney, I have chosen to ignore this issue for a variety of reasons. First, although there is no doubt that large collections of persistent data will exist, it also is clear that as the size of memory increases, more and more important databases will fit in main memory. Thus, the techniques developed here are directly useful for a growing number of applications. Second, if it is impossible to achieve performance that is competitive with explicit techniques for the simpler problem Sidney solves, it seems even less likely that the more challenging problem will allow the use of implicit techniques. Finally, I expect the techniques developed here to be useful in the implementation of systems that support large heaps. In particular, it seems very likely that concurrent GC will be required. Since there is already significant work on many other aspects of the large heap problem, it also only makes sense to attack some of the issues that have received less attention, in particular non-disruptive GC.

In systems where the entire persistent heap cannot fit in memory, memory must be managed as a cache for data residing on disk. Since memory is a cache, persistent data may be mapped into different addresses at different times during execution, implying that the in-memory value of pointers cannot be used on disk. Instead, systems typically use a disk-based identifier often called a persistent ID or PID. However, in the in-memory cache, efficiency argues that inter-object references be native machine pointers. Thus when bringing data into memory, systems convert or *swizzle* PIDs into pointers, and when data is moved back to disk they unswizzle the pointers back into PIDs. Many techniques have been proposed for implementing swizzling [39, 58, 60], but the details are not important here.

To support pointer swizzling, systems need to be able to detect when a reference is made to a PID so that the pointer can be swizzled and the data brought into memory. To detect such references, a general technique known as a *read-barrier* is used. Read-barriers are also used in some garbage collectors and in software-based distributed shared memories to detect references that require system intervention.

In his dissertation [24], Hosking studied the performance issues relating to the implementation of read-barriers for persistence implementations. His basic conclusion is that read-barriers can be made efficient enough to allow even very fine grained persistent data to be managed without undue expense.

Hosking's dissertation complements this one. He studies the cost of read-barriers, which I ignore because Sidney maps the entire persistent heap into memory. He studies write barriers which I use but do not study carefully, in part because of his prior work. On the other hand, he ignores issues of garbage collection and fast commit, on which my work focuses. Our meta-approaches are also complementary: he studies a variety of implicit techniques to establish which is to be preferred, while I focus on showing that specific implicit techniques are competitive with explicit techniques.

11.1.2 Transactions

Transactions were first developed by the database community [22] and in that context include facilities for concurrency control and support for abort as well as support for permanence. The two main low-level implementation techniques for supporting persistence explored by this community are shadow-paging and logging [22]. There is a clear consensus that of these two, logging-based techniques provide the best performance. One of the most attractive aspects of Sidney's design is that it builds on top of a low-level transaction mechanism (in practice, RVM, although other systems could be used as well) that provides recoverable arrays of bytes cached in volatile memory. This factoring of the design insulates Sidney from the lowest level issues of providing fast disk access and recoverability. The design of Sidney can thus focus on providing the abstraction of a garbage-collected heap that is stable and updated atomically. Gray [22] is also a good reference concerning the lower-level issues.

11.2 Mechanisms

In this section, I consider work related to each of the major mechanisms used by Sidney, write logging, garbage collection, and orthogonal persistence.

11.2.1 Implicit Write Logging

A variety of systems require some kind of write barrier as part of their implementation. In addition to persistence systems, these include generational garbage collectors [33, 56] And software-based distributed shared memory systems [12, 56]. These techniques fall into two broad categories: those that record writes on an object level, and those that record writes based on ranges of addresses, ignoring object boundaries. Sidney's write log is a simple form of an object-based technique.

Before discussing the key related work, I should remind the reader that the choice of technique used in Sidney was primarily based on implementation expediency. The SML/NJ compiler already emitted code to create a store-list to be used as a write barrier for its generational collector. I simply enhanced this implementation to record all writes and to log old values. I have made no effort to improve the performance of the current technique or to experiment with other techniques. There should be no significant problem in using range-based techniques in Sidney and no problem at all in using other object-based techniques.

Hosking

Hosking [25, 24, 26] has studied a variety of different techniques for implementing write barriers for persistence systems, both object-based and range-based. SML/NJ's store-list is a simple version of what Hosking calls remembered sets. The techniques he considers include techniques that use in-line code to log writes as well as virtual memory protection based techniques. He conducts a careful comparative study of the various techniques using a persistent Smalltalk implementation and a version of the OO1 benchmark. He concludes that write barriers can be implemented in a manner that allows the efficient management of small persistent objects. The suitability of range-based techniques suffers when the size of the range becomes too large, arguing that page-based methods may be less than ideal, especially since the current trend is for page sizes to increase substantially. Given the high overhead of logging any write to RVM, whether explicit or implicit logging is used, his results strongly support the idea that implicit write logging has performance competitive with explicit logging.

11.2.2 Garbage Collection

The literature on garbage collection is extensive. Wilson wrote a useful survey of uniprocessor GC techniques [59], including incremental (co-routining) and concurrent (preemptive scheduling) collectors of various kinds. Here, I consider only techniques used in previous designs of concurrent collectors for persistent heaps. There have been two classes of such collectors. One class are the copying-based collectors for transactional heaps, using Baker's algorithm for concurrent copying collection. The second class is a single mark-and-sweep collector for persistent, but not transactional heaps. I also discuss some work relating to evaluating the performance of GC.

Baker's Algorithm

Baker's algorithm was the first algorithm for concurrent copying collection and it remains the most popular [6]. I first discuss the general technique and then a version of it that uses virtual memory protection to provide a read-barrier.

Two interrelated differences distinguish Baker's algorithm from replicating collection. First, in collectors based on Baker's algorithm, logically the flip occurs when the client initiates a collection. Second, once the flip has occurred, the client is only allowed to access to-space values. This requirement is known as the *to-space invariant*. The to-space invariant means that there is no need for a consistency mechanism, since the client manipulates only the data that will remain when the collection terminates. The to-space invariant also means that the garbage collector is allowed to destroy the from-space version of an object, after or during copying it to to-space.

The key question in implementing Baker's algorithm is how to enforce the to-space invariant when some of the data has not yet been copied to to-space. Baker's algorithm solves this problem by requiring that all client references go through a read barrier. The read barrier examines each user reference. If the reference is to a to-space value then it is allowed to proceed. If the reference is to a from-space value, that value is copied to to-space and the reference is done to the to-space version. Collectors based on Baker's algorithm also arrange for copying to be done before user references require it. When all the data are copied, the collection terminates and from-space can be reused.

A variety of techniques have been used to implement the read-barrier required for Baker's algorithm. The most obvious technique is to do the test directly, just as Sidney implements its write barrier directly. Unfortunately, references are much more frequent than writes so software-based techniques have high overhead and poor performance. On microcoded machines, the read barrier has been implemented in micro-code. This technique is not compatible with current machine designs, which are not microcoded, and although they result in fewer instructions to implement the write barrier, they may well not result in fewer machine cycles.

Ellis, Li, and Appel [3, 21] proposed a popular version of Baker's algorithm based on virtual memory protection. The read barrier is implemented by using virtual memory protection to protect from-space access when the collector flips. When the client tries to access a from-space object, a fault occurs and the access is redirected to the to-space version of the object. If no to-space version exists one is created by copying the from-space version.

The advantage of this technique is that the access check is done efficiently as part of virtual memory translation. There are a number of disadvantages. First, not all operating systems support the required virtual memory facilities, including the most popular OSs found on personal computers. Even on operating systems that provide the required support, it may be very expensive and hard to implement. In general, handling virtual memory faults is usually not easy or light-weight, especially from a runtime system running at user level. Finally, since at the beginning of the collection very few objects have been copied, the technique often has a large flurry of faults just after the flip. These often prove as disruptive as a single longer pause.

Baker's Algorithm for Persistent Data

There exist two previous designs for concurrent copying collectors for transactional heaps. One was the basis of Dave Detlefs' PhD thesis [19] at CMU and the other was the basis of Eliot Kolodner's PhD thesis [30] at MIT. Although the two techniques differ in many details, at the level discussed here they are quite similar, so I discuss them together. Essentially both attempt to extend VM-based Baker's algorithm collection to work on persistent heaps.

The to-space invariant makes Baker's Algorithm more complicated to implement when the system must also support transactions. Since the client begins to manipulate to-space objects as soon as the collection begins, the transaction system, and particularly the recovery mechanism, must be aware of both from-space and to-space objects simultaneously. In contrast, Sidney defers the need to manipulate the to-space objects until a flip. To the recovery system it is as if it is just suddenly dealing with a whole new set of objects, not copies of the ones it was using before.

Since the recovery system must deal with a mixture of to-space and from-space objects and because the system may crash at any point during the collection, both Detlefs and Kolodner found it necessary to make the collection process itself atomic. Making the collection atomic required making both the copy and scan steps atomic, not just the flip as Sidney requires. Making these steps atomic causes the details of the collection and recovery algorithms to become tightly intertwined. Since both GC and recovery algorithms are complex, intertwining them makes for an especially complex design. Both Detlefs' and Kolodner's dissertations devote significant effort to specifying their algorithms and arguing for their correctness. I believe that although both of them spent significant time attempting an implementation, neither implementation was completed because of this complexity.

It is difficult to speculate about the performance of these two approaches. However, making copying and scanning atomic requires that information be written to the log and eventually to the disk for each operation. Both Detlefs and Kolodner suggest optimizations for this log writing, but in the end there would be more log traffic. As we have seen log traffic is expensive, so it seems likely that their techniques would prove more expensive than Sidney's design.

One apparent advantage of these techniques is that recovering from a crash allows a GC to restart in the middle, while for Sidney any GC work is lost. It is not hard to extend Sidney to allow the GC state itself to be check-pointed, thus achieving the same result. Since recovery of the collector is not tied to the correctness of the system for client operations, this can be done more flexibly and at varying grain size than for the techniques of Detlefs and Kolodner.

Almes

One previous concurrent collector for persistent data has been implemented by Guy Almes for the Hydra system [1]. Hydra did not support transactions or orthogonal persistence, and it is not clear exactly what guarantees it offered to the client about data consistency. The collector itself was based on mark-and-sweep collection, which, since it does not move objects, is somewhat easier to make concurrent than copying collection. Taken together, the lack of transactions and the different collection strategy make Almes' design much more like a collector for a transitory heap than either Sidney or Detlefs' or Kolodner's designs. Due to the rather unique hardware environment of Hydra, it is impossible to make any valid performance comparisons between Almes' work and Sidney.

GC Performance

The work presented here is not only the first to implement concurrent collection of transactional heaps, but also to date the only significant performance study of many aspects of persistent heaps.

Several aspects of the performance evaluation are novel. There are no other detailed measurements of transactional garbage collector implementations, even for stop-the-world collectors. The closest work of this sort is that by Zorn [17], in which he uses traces to drive a simulation of various partitioning techniques for persistent heaps. This work does not study an actual implementation, but rather probes the effects of a potentially important policy decision.

The use of tracing to drive a comparison of GC and malloc-and-free is also novel, even for transitory heaps. However, Zorn has done a more convincing studying of a more limited collection technique [63].

By using a conservative collector that was compatible with the interface to malloc-and-free, Zorn was able to compare the performance of GC to that of a variety of malloc-and-free implementations. His study shows that for transitory data, GC has CPU costs that are competitive with those of malloc-and-free, although it takes more space and has somewhat worse locality. A related study by Detlefs, Dosser, and Zorn [20] shows similar results for some large C++ programs.

These studies are more convincing than the one here in that they study the effects in context using real applications, while my trace-driven approach clearly has some artifacts caused by its trace-driven nature. These artifacts favor malloc-and-free, so the current results are conservative. However they also prevent a careful study of GC pause times using the same programs as for malloc-and-free. However, the studies requiring the use of conservative collectors are not as flexible as my technique. In particular, the conservative collectors used in some of these studies do not include some key performance enhancements, such as generational collection. It is expected that these studies represent a worst case and that real collectors may perform much better in practice.

11.2.3 Orthogonal Persistence

The first implementations of orthogonal persistence were for PS-Algol [15]. They used techniques based on GC as does Sidney. However, their original model was that a commit also terminates the process, so there was no transitory heap remaining after commit, eliminating the possibility of incorrect transitory heap pointers.

Argus and Kolodner

The previous system that most closely resembles Sidney is Kolodner's, for the Argus programming language [34]. Argus uses orthogonal persistence, in the sense that persistence is determined by reachability. Argus also may identify persistent objects by type. Kolodner is concerned about providing good commit performance and also proposes to use separate heaps for persistent and transitory data.

Like Sidney, Argus must identify newly persistent data and guarantee that it is stable on commit. How this is done is quite different from Sidney. When a persistent object is updated, before the update may complete, a "Concurrent Stability Tracker" is invoked that finds all of the reachable data and logs the newly persistent data to stable storage. This technique is needed in part because Kolodner follows a strict write ahead log protocol.

I believe Sidney's technique is superior. Although in the current implementation moving data into the persistent heap is done as lazily as possible, it could be made more eager and it is compatible with a range of logging protocols. By deferring the update, Sidney does not need to identify persistent objects in the write barrier and can allow the write to complete quickly (in a few instructions). Although not supported currently, Sidney could even move the newly persistent data concurrently with client activity. Also being lazy rather than eager means that there is the possibility of piggybacking the transfer to the persistent heap on a garbage collection that must occur in any case. In general, it seems desirable to take advantage of the fact that newly persistent data only needs to be stable at the end of commit.

Kolodner also the needs to update the pointers that refer to objects that have been moved from the transitory to persistent heap. His solution is to force all reads to check for forwarding pointers. I do not believe that this approach has acceptable performance characteristics, since it substantially increases the cost of every pointer dereference. Kolodner offers various suggestions for reducing these costs, but I believe they are best avoided, rather than reduced. Forcing reads to detect when an object has been moved to the persistent store is also the technique used by other systems [38], but no other system has an implementation of commit with as favorable asymptotic time bounds as Sidney.

Performance

Hosking studies the performance implications of some of the overheads due to orthogonal persistence, in particular, as mentioned, the cost of the write barrier needed to detect newly persistent data and the read-barrier needed by his system to detect when a reference is made to an object that has been moved. The current study is the first to look at the actual costs of moving data into the persistent heap and to show that it is negligible compared to costs shared by systems that do not implement orthogonal persistence.

Chapter 12

Conclusions

The goal of this dissertation was to confirm my thesis: a persistent storage manager based on implicit techniques, such as garbage collection, has greater safety and can have performance that is competitive with a persistent storage manager based on explicit techniques, such as malloc-and-free. To demonstrate the thesis, I designed and implemented a persistent storage manager, Sidney, based on implicit techniques. I then compared the performance of Sidney both to explicit techniques and to competing implicit techniques. The most significant conclusion of this effort is that my results strongly support the thesis.

In the sections that follow, I first present a detailed list of the contributions of the dissertation, complementing the high-level list of contributions found in Chapter 1. I then summarize the dissertation with an emphasis on the points that support the thesis. I then provide a series of recommendations for others who wish to build persistent storage managers. The dissertation concludes with a discussion of possible future work and a few words of personal retrospective.

12.1 Contributions

The contributions of this dissertation can be categorized into those for the design and implementation and those for performance evaluation.

For the design and implementation, the contributions include:

- The design and implementation of a new concurrent garbage collection algorithm: replicating collection.
- The first implementation of a concurrent garbage collector for a transactional persistent heap and of a concurrent copying garbage collector for a persistent heap of any kind.
- The first design of a concurrent copying garbage collector for a persistent heap that uses a from-space invariant and does not require low-level coordination between the collector and transaction system or between the collector and the client.

- The first design and implementation of orthogonal persistence that supports transactions and has asymptotic complexity that does not either depend on the amount of transitory heap data or require a read-barrier to detect objects that have been made persistent.

For the performance evaluation, the contributions are:

- The first demonstration that a system supporting persistence using implicit storage management techniques can have performance competitive with systems using explicit techniques.
- The first detailed performance study of a garbage collector for a persistent heap.
- A framework and suite of benchmarks for comparing explicit persistent storage management to implicit persistent storage management and a comparative study of explicit persistent storage management and implicit persistent storage management using that framework.
- The first trace-based comparison of malloc-and-free and garbage collection.
- The first demonstration of speedup from concurrent collection due to overlapping I/O with computation.

12.2 Summary of Results

The need for safe management of persistent data argues strongly that implicit techniques should be used to support transactions and other memory management functions. Sidney uses three main implicit techniques: automatic logging of writes, garbage collection, and orthogonal persistence. Automatic logging of writes prevents the application programmer from forgetting to record a write with the transaction system, thus preventing lost updates to persistent data. Garbage collection prevents the programmer both from forgetting to free storage and from freeing storage too soon, thus preventing storage leaks and dangling pointers. Orthogonal persistence prevents the programmer from forgetting to make some important datum persistent. These features provide safer storage management than explicit solutions to similar problems because they prevent the programmer from making common errors that threaten the integrity of persistent data. This property provides the basis for the safety claims of the thesis.

The need for good performance argues that implicit techniques should only be used if they do not compromise the application programmer's ability to achieve their performance goals. The design chapters show how each implicit feature, write logging, garbage collection, and orthogonal persistence, can be implemented with asymptotic performance that is either equal to or incomparable with explicit techniques. The implementation chapter shows how this design is realized in practice. Together these parts of the dissertation show that performance competitive with explicit techniques is not impossible to achieve.

The need for good performance requires not only that implicit techniques have good asymptotic performance, but also acceptable performance in situations that occur in practice. The performance evaluation establishes this fact. The evaluation proceeded along two axes: one comparing Sidney to explicit techniques and to other implicit techniques, and the other examining Sidney's performance in terms of throughput and in terms of latency. Four benchmarks were used in the study, **trans** and **coda**, which are used to compare Sidney to explicit techniques and which focus on throughput, and **comp** and **OO1**, which are used to compare Sidney to other implicit techniques and which consider both throughput and latency. I now review the main performance results in terms of how each implicit feature fares.

The cost of write logging is not considered in any detail in the performance evaluation. There are two reasons for this omission. First, the write logging done by Sidney is an example of a general class of techniques called write barriers. The cost of write barriers have been studied extensively [12, 56], and in particular by Hosking [24]. I choose not to duplicate this work in a more limited context. Second, even explicit techniques must record writes to persistent data. My argument is not that support for transactions is cheap (it is not), but rather that implicit techniques have no fundamental performance disadvantage compared to explicit ones. Since write logging is required in both cases, a careful understanding of its cost is not needed to support my argument. The discussion of future work (Section 12.4) argues that some small changes to RVM may make implicit logging even cheaper than explicit logging.

The cost of garbage collection is the focus of much of the performance evaluation. Comparing the throughput of concurrent collection to a malloc-and-free used in a production system, I found that GC had a significant advantage when managing persistent data. There were two basic reasons for this advantage. First, the use of GC allowed the costs of allocating persistent storage to be greatly reduced because the operations that needed to be recorded by RVM could be batched and many of them eliminated. GC allowed other operations to be batched and eliminated as well. Overall GC needed to do less I/O and use less CPU time than malloc-and-free. The other advantage was that the cost of reclaiming free storage can be made asynchronous and much of it overlapped with user computation. This latter point held even on uniprocessors, because much of the work that could be overlapped was I/O.

I also compared concurrent collection to stop-and-copy collection. Not surprisingly, there were throughput advantages there as well. These stemmed from the ability to overlap GC work with the client, in particular, overlapping I/O with computation was a significant contributor. I also compared the throughput of concurrent collection to that of stop-and-copy collection as the livesize in the heap increased. This comparison showed that the advantages of concurrent GC increase as the size of heaps increase.

The throughput advantages of concurrent collection are important, but the main reason to introduce the technique is to help control the latency introduced by GC pauses. I used both **comp** and **OO1** to investigate the pauses due to GC and `commit`. Concurrent collection resulted in improvements in pause times compared to stop-and-copy collection by at least an order of magnitude, and often more. Furthermore, the pauses caused by the user doing `commits` were as long or longer than those caused by concurrent collection and they are much more frequent, despite that for benchmarking purposes the collection frequency is much greater than it would be in practice. The pauses caused by concurrent collection are also independent of the amount of persistent data in the heap, while the length of pauses due to stop-and-copy collection increase linearly with the amount of data in the heap.

Unfortunately, despite pause time performance that is dramatically better than stop-and-copy collection and less disruptive than user `commits`, the pause times, especially for **OO1**, are still longer than I would prefer. This problem is the most serious weakness in support for the thesis. The performance evaluation identifies the current bottleneck to be a simplistic choice of algorithm for redirecting transient heap pointers into the persistent heap when the heap flips. I expect that when one of several possible more sophisticated solutions is applied to this problem, these pause times will improve significantly. Also, since the flip does not require any I/O, we can expect that using more modern processors will improve these times significantly as well.

12.3 Recommendations

In this section, I discuss my recommendations to others who might wish to implement their own general-purpose persistence system. One might view this as answering the question, "What would I do differently if I could?" I have tried to answer the question somewhat more broadly than this, in particular in terms of the programming language issues. I have reserved my comments for what might lie ahead for Sidney for the future work.

Probably the most obvious question is whether to support general-purpose persistence at all. My answer is "yes". Supporting general-purpose persistence allows the application programmer to manipulate permanent data almost as easily as transitory data, rather than forcing the programmer to force the data to fit a less flexible model of persistence like files or databases. One significant advantage is that fine-grained changes to persistent data can be made atomically and relatively efficiently, which is difficult with more special-purpose persistence systems, like files, except when manipulating the exact data types they support.

The next significant question is whether to support transactions. Again, my answer is "yes". Transactions are a well-understood and widely-used abstraction to support fault-tolerance. The only obvious alternative is some kind of transparent checkpointing. This approach is useful when trying to make applications fault-tolerant transparently, but it does not provide the same kinds of guarantees to the application programmer that transactions do.

The next feature to consider is implicit logging of writes. My answer is "yes" if the programming language (and its implementation) make it feasible. SML is particularly well suited to implicit logging, since its model of computation emphasizes binding rather than mutation. It would not be feasible to log every assignment in, for example, a C program. This is not a fatal problem however. All that is really required is to log writes to persistent data. These writes can be identified in a variety of ways. First, writes to stack-allocated data need not be logged and it is reasonable to expect that the compiler can often distinguish between stack writes and heap writes. Second, persistent data can be in unique parts of the address space and a quick check can avoid logging all other writes. This situation is also one where using virtual memory protection might be a profitable technique. Finally, if we make persistence an explicit property of data, like type, then the compiler can explicitly distinguish and emit code for just the stores to persistent data.

The next feature to consider is garbage collection. Again, the question of programming language and language implementation arises. Certainly if the language allows it, I believe strongly that garbage collection should be used. GC provides critical safety benefits and also makes the programmer's task easier as well. Sidney is based on copying garbage collection, which puts the greatest burden on the language, since it must allow data to be moved. However mark-and-sweep collection can be used in more settings and "conservative" collectors based on both copying and mark-and-sweep have been used with a wide variety of languages, including C. The possibility of concurrent mark-and-sweep collection of persistent heaps is discussed in the future work. If these techniques prove feasible and compatible with conservative techniques, I believe they are the technique of choice.

The final feature to consider is orthogonal persistence. Here the issue is as much one of language design as language implementation. At least for languages that support copying collection, the techniques described here show that orthogonal persistence can be implemented with acceptable performance. For some languages, like SML or the Lisp family, the programmer expects an abstract view of storage and orthogonal persistence seems very fitting in these languages. In languages like C, C++ or even Modula-3, it seems that tying persistence to data type or other explicit properties may be more natural and provide the user with greater control, although at the risk of possibly allowing the programmer to fail to make critical data persistent. I suspect that implementation issues will dominate in language implementations that do not already support copying collection.

12.4 Future Work

This section concludes the dissertation by discussing some possible future work. The section begins by looking at possible changes to RVM. It then looks examines enhancements to Sidney specifically and finally concludes with some more general work that relates to replicating collection and persistence in general.

12.4.1 RVM

RVM places no particular restrictions on Sidney's functionality, but it does have a significant impact on its performance. Thus, one obvious way to improve Sidney's performance is to improve RVM. Some of these improvements would benefit other RVM clients, while others are likely to improve only Sidney or a similar system. I look first at general improvements and then at some Sidney-specific ones.

RVM does extensive error checking of arguments, consistency of data structures, etc. This error checking is important when developing new code using RVM and probably should also remain in place when RVM is used directly by the application programmer. For Sidney and other systems that layer a new interface over RVM, it seems desirable to allow these checks to be turned off, preferably at compile time. That Sidney has not raised an RVM exception for several years seems good evidence for this.

Another related enhancement would be to provide an interface that allows multiple writes to be recorded with one call to RVM. This change would allow updates to amortize both error checking and other setup operations over a more substantial number of operations. Neither of these enhancements would be difficult to add to RVM.

A more substantial change to RVM would be support for *write ahead logging* (WAL). In the current RVM implementation all logging to the disk is done on demand when the user ends an RVM transaction. The database literature suggests that by being eager about logging data to the disk, the performance of long running transaction can be substantially improved. I will discuss the possible impact of this change on Sidney in the sections that follow.

One change that is likely to benefit most RVM clients, but not Sidney, would be incremental log truncation. Currently, RVM waits until its log fills to some high-water mark and then truncates the log in a batch fashion, using the same mechanism as used for recovery. Truncation is an expensive operation and it seems likely that a more incremental approach could benefit many clients. This advantage is less likely to be true for Sidney, because when Sidney performs a GC, it creates a new complete checkpoint of the heap on disk. Thus instead of truncating the log, Sidney can just throw it away. As long as this happens before the log needs to be truncated, the cost of truncation will not be important to Sidney.

For the last point to hold, there is one feature that RVM would need that would benefit Sidney, but probably not most RVM clients. As discussed in the performance evaluation, a substantial part of the cost of Sidney heap image to disk is performing a log truncation when flipping. To enable the previous optimization and eliminate the need for truncation, we can enhance RVM by adding the ability to just drop log records that apply to certain regions of memory. This change should be easy and it should have a substantial impact on Sidney's performance.

12.4.2 Sidney

The most obvious improvement to Sidney would be to implement one of the pause time improvements mentioned in Section 9.2.2. This change would eliminate the current source of Sidney's longest pauses and hopefully would allow Sidney to have pause times that are not dependent on the size of the transitory heap. I expect to perform this enhancement before publishing this work further.

Another way to improve Sidney's performance will be to use faster machines. The machines used to study Sidney here, while current when this work began, now are obsolete. I have a new SGI multiprocessor, which has CPU performance approximately 8 times (based on comparing SPECINT-92 results) that of the one used in this study. It also has improved I/O performance, although the disks rotate only somewhat faster, 7200 RPM. I expect to port Sidney to this machine soon. This port will allow me to compare Sidney's performance on a much faster machine to the current results. Since CPU speeds are likely to continue to increase faster that I/O latencies, any changes found should be good predictors of future trends as well. In general, as I/O becomes a more dominant factor in an application's performance I expect Sidney's advantages, many of which derive from overlapping I/O with computation, to increase.

I am confident that these two changes, eliminating the current bottleneck and using a modern machine, will allow Sidney's pause times to drop well below those for disk access and virtual memory. At that point there will be little point in trying to achieve further improvements.

A more far reaching change would be to add WAL to Sidney. Currently, Sidney is lazy about moving data into the persistent heap. Data is allocated in new-space, then copied into old-space, and finally copied into the persistent heap. It would be possible (even without WAL) to copy data directly from new-space to the persistent heap, and this would fit nicely with WAL. Recording writes to the persistent heap eagerly using WAL should provide the same basic benefits that it does to databases. These changes should improve the performance of long-running transactions.

12.4.3 More General Future Work

This part concludes the dissertation with a look at some possible future work of a more far reaching kind. I do not mean to imply that this work is less likely to be done, just that it focuses less closely on the current system.

Really Real-Time Collection

Replicating collection allows one to limit pause duration and thus make GC much less disruptive than stop-and-copy collection. Bounded pause time is the sense that the garbage collection community uses when they refer to real-time collection. However, bounding pause times is not sufficient for hard real-time systems. The fundamental problem is that replicating GC must interrupt all user threads and force them into a barrier synchronization when flipping. Hard real-time demands that there be no barrier synchronization and that the client threads choose when to synchronize with the collector. Hard real-time systems are able to guarantee that these synchronization attempts occur with a certain frequency, which is necessary if any algorithm that does not interrupt the client is to be able to make guarantees of forward progress.

Currently, no purely copying collector running on stock hardware satisfies the demands of the hard real-time community. No explicit storage allocator does either, so this community is shackled with static storage allocation or at best a special-purpose allocator. Jim O'Toole and I have looked at this problem briefly and believe that replicating collection can be adapted for use in hard real-time applications. This work is preliminary enough that it would be premature to discuss it here.

Flash Memory

In Sidney, the rate at which small transactions can `commit` is limited by the rotational delays of the disks used to provide stable storage. Even if these devices were significantly faster, the cost of using RVM, and ultimately the cost of system calls to do I/O, limits this rate. A new memory technology, flash memory, offers a potential solution to both of these problems, at a cost that promises to be much more attractive than competing solutions like battery-backed DRAM.

Flash memory is based on the same basic technology as EEPROM and is fundamentally different from conventional DRAM and SRAM. Flash has the following attractive features: it is persistent and requires no power to maintain its state even for long periods of time; it can be read at the same speed as DRAM; it has density and thus cost equivalent to DRAM; and finally flash is a commercially viable technology now producing revenues in the billions of dollars. It is possible that flash will ultimately have density and thus cost significantly below those of DRAM. In particular, Intel has laboratory examples of flash cells that can store two bits of information rather than one. Naturally, flash also has some significant disadvantages: it can only be written once before it must be erased; it must be erased in large blocks, not on a word by word basis; writes are about one hundred times slower than reads; erases are slower still; and finally, it wears out. Current flash memories can only sustain about 100,000 write-erase cycles. Some of these limitations may be relaxed as flash develops further, in particular the number of write-erase cycles is expected to increase substantially over time.

In work initially suggested by Morgan Price, we are investigating using flash memory to provide significant improvements in the performance of systems like Sidney. These improvements will come from two sources. First, although writes to flash are slower than reads, they are still much faster than writes to disk, and they do not have the same locality demands that fast writes to disk do. Thus flash will significantly improve the speed of synchronous writes. Once the problem of the speed of synchronous writes is reduced, the system will be limited by the CPU costs of doing I/O. Flash also allows a solution to this problem. Instead of using flash as an I/O device, as has mostly been done so far, we propose to integrate flash into the memory hierarchy and to allow the user to access it directly. This architecture will allow reads to proceed at normal speeds and will make it possible to minimize the overhead of writes, allowing as much performance to be achieved as possible. The properties of copying GC and SML mesh well with flash. Most of the data in an SML heap are immutable and thus can be stored in flash just as well as DRAM. By doing allocation of all data in DRAM and then copying the immutable data to flash and leaving the mutable data in DRAM, we can take advantage of fast allocation in DRAM and minimize the overall amount of DRAM needed. By using flash to provide a log similar to that of RVMs disk log, we can provide recoverability for the mutable data as well. Copying GC is attractive because when it copies data, it leaves large segments of memory free, which is ideal for the erase requirements of flash.

One nice aspect of this proposed work is that it can leverage off the current work to provide a convincing comparative experiment for understanding the advantages and disadvantages of flash. This demonstration would consist of the following: the existing system, based on disk technology; an enhanced version of RVM that uses flash as a fast disk, showing how flash improves the performance of conventional I/O-based designs; and finally a reimplementation of the current interface, using flash directly, showing the effects of adding flash to the memory hierarchy. The current benchmark suite and benchmarking infrastructure should provide a good basis for comparing these three systems.

Mark-and-Sweep Collection

Another future work moves beyond copying garbage collection to consider adding persistence to a system based on mark-and-sweep collection. Since mark-and-sweep collectors do not move data, it should be substantially easier to provide concurrent collection that interacts well with transactions.

One interesting language implementation that uses a mark-and-sweep collector is the Java implementation from Sun Microsystems. I plan to add persistence to Java and then to explore building a concurrent mark-and-sweep collector for the system.

12.5 Final Words

This dissertation represents the culmination of my formal training in Computer Science. It represents my interests in three significant and interrelated areas of the field, Programming Languages, Systems, and Experimental Computer Science.

The initial design of Sidney primarily reflects my interests in Programming Languages. The design emphasizes the safe and natural integration of persistence into a modern high level programming language. The use of write logging, garbage collection, and orthogonal persistence reflects a programming language designer's concern for safety, elegance and ease of programming. These concerns are fundamental to Sidney.

The implementation of Sidney primarily reflects my interests in Systems. At least some of the systems community holds that a slow system is no more useful than a slightly incorrect one. My goal in implementing Sidney was to build a system that had good performance. To achieve this goal required both innovative work on new techniques, especially for garbage collection, and careful attention to systems issues, such as avoiding synchronous disk writes. These concerns are also fundamental to Sidney.

The final phase of my work on Sidney primarily reflects my interests in Experimental Computer Science, as applied to both Programming Languages and Systems. My goal was to validate the claim that safe persistent storage management could be achieved without compromising on performance. The methodology was the classic experimental one: I conducted a series of experiments in which I compared the results obtained with Sidney to those obtained with a control, either an explicit technique or a more basic implicit one. I paid careful attention to experimental design and setup and to data analysis and presentation. Although not fundamental to Sidney itself, these concerns are fundamental to the demonstration of the thesis.

By combining these three approaches, I have shown that it is possible to design and build a persistent storage management system that is both safe and efficient.

Bibliography

[1] G. T. Almes.
Garbage Collection in a Object-Oriented System.
Technical Report CMU-CS-80-128, Carnegie Mellon University, June 1980.

[2] A. Appel.
Simple Generational Garbage Collection and Fast Allocation.
Software–Practice and Experience, 19(2):171–183, February 1989.

[3] A. W. Appel, J. R. Ellis, and K. Li.
Real-time Concurrent Garbage Collection on Stock Multiprocessors.
In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, 1988.

[4] A. W. Appel and D. B. MacQueen.
A Standard ML Compiler.
In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.
Volume 274 of *Lecture Notes in Computer Science*.

[5] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison.
An Approach to Persistent Programming.
Computer Journal, 26(4):360–365, December 1983.

[6] H. G. Baker.
List Processing in Real Time on a Serial Computer.
Communications of the ACM, 21(4):280–294, 1978.

[7] R. V. Baron, D. L. Black, W. Bolosky, J. Chew, D. B. Golub, R. F. Rashid, A. Tevanian Jr, and M. W. Young.
Mach Kernel Interface Manual.
School of Computer Science, Carnegie Mellon University, February 1987.

[8] H. J. Boehm and M. Weiser.
Garbage Collection in an Uncooperative Environment.
Software Practice and Experience, 18(9):807–820, September 1988.

[9] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams.
The GemStone Data Management System, chapter 12.
ACM Press, 1989.

[10] S. K. Card, T. P. Moran, and A. Newell.
The Psychology of Human-Computer Interaction.
Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[11] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Skekita, and S. L. Vandenberg.
The EXODUS Extensible DBMS Project: An Overview.
In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan-Kaufmann, 1990.
also available as WISC-CS-TR 808.

[12] J. B. Carter, J. K. Bennett, and W. Zwaenepoel.
Implementation and Performance of Munin.
In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164. ACM, October 1991.

[13] R. G. G. Cattell.
An Engineering Database Benchmark.
In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 247–281. Morgan-Kaufmann, 1991.

[14] T. W. Christopher.
Reference Count Garbage Collection.
Software Practice and Experience, 14(6):503–507, June 1984.

[15] W. Cockshott, M. Atkinson, K. Chisholm, P. Bailey, and R. Morrison.
Persistent Object Management System.
Software Practice and Experience, 14(1):49–71, January 1984.

[16] G. E. Collins.
A Method for Overlapping and Erasure of Lists.
Communications of the ACM, 2(12):655–657, December 1960.

[17] J. E. Cook, A. L. Wolf, and B. G. Zorn.
Partition Selection Policies in Object Database Garbage Collection.
In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 371–382, Minneapolis, MN, May 1994.

[18] E. Cooper, S. Nettles, and I. Subramanian.
Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management.
In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 43–52, June 1992.

[19] D. Detlefs.
Concurrent, Atomic Garbage Collection.
Technical Report CMU-CS-TR-90-177, Carnegie Mellon School of Computer Science, October 1990.

[20] D. Detlefs, A. Dosser, and B. Zorn.
Memory Allocation Costs in Large C and C++ Programs.
Computer Science Technical Report CU-CS-665-93, Digital Equipment Corporation and University of Colorado, August 1993.

[21] J. R. Ellis, K. Li, and A. W. Appel.
Real-time Concurrent Garbage Collection on Stock Multiprocessors.
Technical Report DEC-SRC-TR-25, DEC Systems Research Center, February 1988.

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

[23] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing.
Composing First-Class Transactions.
ACM Trans. on Prog. Lang. and Systems, Short Communications, 16(6):1719–1736, 1994.

[24] A. L. Hosking.
Lightweight Support for Fine-Grained Persistence on Stock Hardware.
Technical Report 95-02, Computer Science Department, University of Massachusetts at Amherst, February 1995.

[25] A. L. Hosking and J. E. B. Moss.
Object Fault Handling for Persistent Programming Languages: A Performance Evaluation.
In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '93) Proceedings*, pages 288–303. ACM Press, 1993.
Published as *ACM SIGPLAN* Notices 28(10), October 1993.

[26] A. L. Hosking, J. E. B. Moss, and D. Stefanović.
A Comparative Performance Evaluation of Write Barrier Implementations.
In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 92–109. ACM Press, October 1992.
Published as *SIGPLAN Notices 27*(10), October 1992.

[27] P. Hudak et al.
Report on the Programming Language HASKELL.
Technical Report YALEU/DCS/RR-777, Yale University, CS Dept., April 1990.

[28] R. L. Hudson and J. E. B. Moss.
Incremental Collection of Mature Objects.
In Yves Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 388–403. Springer-Verlag, September 1992.

[29] R. Jain.
The Art of Computer Systems Performance Analysis.
Wiley, New York, 1991.

[30] E. K. Kolodner.
Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap.
Technical Report MIT/LCS/TR-534, Massachusetts Institute of Technology, February 1992.

[31] P. Kumar.
Personal communication.

[32] R. J. Larson and M. L. Marx.
An Introduction to Mathematical Statistics and its Applications.
Prentice-Hall, Englewood Cliffs, NJ, 1986.

[33] H. Lieberman and C. Hewitt.
A Real-Time Garbage Collector Based on the Lifetimes of Objects.
Communications of the ACM, 26:419–429, June 1983.

[34] B. Liskov and R. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Language and Systems, 5(3):382–404, July 1983.

[35] H. Mashburn and M. Satyanarayanan.
RVM User Manual.
School of Computer Science, Carnegie Mellon University, Spring 1992.

[36] R. Milner, M. Tofte, and R. Harper.
The Definition of Standard ML.
MIT Press, 1990.

[37] J. G. Morrisett and A. Tolmach.
Procs and Locks:
A Portable Multiprocessing Platform for Standard ML of New Jersey.
In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego*, pages 198–207, May 1993.

[38] R. Morrison, A. L. Brown, R. Carrick, R. Conner, and A. Dearle.
On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments.
In *Advances in Object-Oriented Database Systems*, pages 334–339, 1988.

[39] J. E. B. Moss.
Working with Persistent Objects: To Swizzle or Not to Swizzle.
COINS TECHNICAL REPORT 90-38, University of Massachusetts, Amherst, MA, May 1990.

[40] G. Nelson, editor.
Systems programming with Modula-3.
Prentice-Hall, 1991.

[41] S. M. Nettles and J. W. O'Toole.
Real-Time Replication Garbage Collection.
In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 217–226. ACM, June 1993.

[42] S. M. Nettles, J. W. O'Toole, D. Pierce, and N. Haines.
Replication-Based Incremental Copying Collection.
In *Proceedings of the SIGPLAN International Workshop on Memory Management*, pages 357–364. ACM, Springer-Verlag, September 1992.

[43] J. O'Toole and S. Nettles.
Concurrent Replicating Garbage Collection.
In *ACM Symposium on LISP and Functional Programming*. ACM Press, June 1994.

[44] J. O'Toole, S. Nettles, and D. Gifford.
Concurrent Compacting Garbage Collection of a Persistent Heap.
In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, December 1993.

[45] J. W. Pratt and J. D. Gibbons.
Concepts of Nonparametric Theory.
Springer-Verlag, New York, 1981.

[46] Richard Greenblatt.
The LISP Machine.
In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*. McGraw Hill, 1984.

[47] J. E. Richardson and M. J. Carey.
Persistence in the E Language: Issues and Implementation.
Software Practice and Experience, 19(12):1115–1150, December 1989.

[48] M. Rosenblum and J. K. Ousterhout.
The Design and Implementation of a Log-Structured File System.
ACM Transactions on Computer Systems, 10(1), February 1992.

[49] P. Rovner.
On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked, Concurrent Language.
Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

[50] M. Satyanarayanan et al.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Trans. Computers, 39(4):447–459, April 1990.

[51] 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.

[52] J. Seligmann and S. Grarup.
Incremental Mature Garbage Collection Using the Train Algorithm.
In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95)*, number 952 in LNCS, pages 235–252, Aarhus, Denmark, August 1995. Springer-Verlag.

[53] A. Z. Spector.
The Design of Camelot.
In *Camelot and Avalon*. Morgan-Kaufmann, 1991.

[54] G. L. Steele Jr and G. J. Sussman.
Scheme: An Interpreter For the Extended Lambda Calculus.
AI Memo 349, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1975.

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

[56] D. Ungar.
Generational Scavenging: A Non-Disruptive High Performance Storage Management Reclamation Algorithm.
In *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 15–167, Pittsburgh, Pennsylvania, April 1984.

[57] Unix System Laboratories.
TUXEDO System Product Overview, February 1993.

[58] S. J. White and D. J. Dewitt.
A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies.
In *18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, October 1992. Morgan-Kaufman.

[59] P. R. Wilson.
Uniprocessor Garbage Collection Techniques.
In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, pages 1–42. ACM, Springer-Verlag, September 1992.

[60] P. R. Wilson and S. V. Kakkad.
Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware.
In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.

[61] P. R. Wilson, M. S. Lam, and T. G. Moher.
Effective Static-Graph Reorganization to Improve Locality in Garbage-Collected Systems.
In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM Press.
Published as *SIGPLAN Notices* 26(6), June 1992.

[62] J. M. Wing.
The Avalon Language.
In *Camelot and Avalon*. Morgan-Kaufmann, 1991.

[63] B. Zorn.
The Measured Cost of Conservative Garbage Collection.
Software—Practice and Experience, 23(7):733–756, July 1993.
also available as CU-CS-573-92.