Memory Management on a Massively Parallel Capability Architecture

Jeremy Hanford Brown
jhbrown@ai.mit.edu

December 3, 1999

Contents

Chapter 1

Introduction

My research group is currently designing a massively parallel distributed-memory capability architecture. This architecture will feature fine-grain hardware-recognized objects, all of which exist in a single, globally-shared address space. Different portions of the address space are striped across the physical processors at different granularities, so an object may either reside entirely upon one processor or striped across a number of the system's processors.

The primary means of long-term storage for this machine will be a persistent heap in which all objects reside. One of our design goals for the overall system is to provide complete pointer safety; this requires system-wide management of object allocation and garbage-collection. Memory management must respect both the limited inter-node network bandwidth and the limited amount of per-node physical memory.

In this document I propose a thesis in which the primary topic is exactly this memory management problem. The project I propose is composed of four parts: the written design of the entire memory-management system; the simulation of the most critical elements of the specified system; the design and implementation of a programming language and benchmark applications needed to drive the simulation for evaluation purposes; and the detailed evaluation of the heap implementation.

1.1 System goals

Memory-management on our architecture has several challenges: an object on one processing node may reference objects on many other nodes; some objects may be striped over many nodes; and the persistent heap may exceed the size of physical memory many times over. We do have the advantage that inter-processor references are very fast (low-latency), but this is also a challenge in that memory management overhead on inter-processor operations must be correspondingly tiny in order not to have terrible performance implications.

In light of these challenges, a few of my specific goals for this project are:

- Total pointer safety: no piece of memory can be accessed by a thread of control that doesn't have legitimate access to it.
- Memory reclamation: garbage data must be eliminated.
- Efficiency of several sorts:

- User-thread memory accesses should experience no runtime overhead. Pointers must generally point straight to their target objects, even when the target is on a different node, since with our low-latency network the cost of extra indirection would be unacceptable.
- Memory management, particularly garbage collection, must not cause excessive paging even when the heap exceeds the size of main memory many times over.
- Memory management, particularly garbage collection, must not consume excessive inter-node communications bandwidth.

I should note that I have a fairly unusual advantage to offset the various challenges I've just listed: because the target computer's architecture is itself a work-in-progress, I can design custom hardware support for the memory-management system.

1.2 Key memory management concepts

The memory-management scheme I propose is primarily based on three key concepts: *areas*, *indirect reference counting*, and *deactivating* idle areas.

An area is composed of a set of addresses which are resident entirely upon one node. An area can be independently garbage-collected, using any pointers into the area from other areas as part of the root set. An object which lives entirely on one node is allocated within a single area; an object which is striped over multiple nodes is allocated within a group of "correlated" areas, one on each node.

Indirect reference counting (IRC) [Piq91] is a reference-counting scheme used to keep track of how many remote areas contain pointers to an object. The compelling feature of IRC is that it requires exactly two inter-area messages for every inter-area pointer-copy. The first message is the actual copy from area A to area B; this makes B a child of A in the dynamically-created copy-tree for the pointed-to object. The second message is sent from B back to area A when local garbage-collection in B discovers both that B no longer has any copies of the pointer, and that B currently has no children in the copy-tree.[1] In my proposed implementation, the copy-tree (AKA the diffusion tree) is represented with IN and OUT sets in each area; the IN set provides the external-root-set for independently GCing an area. I propose special hardware to perform the manipulation of IN and OUT sets in parallel with user-code in the common case.

Note that IRC shares the limitation of all reference-counting schemes of never discovering garbage cycles. While area-local GC will find intra-area cycles, an occasional global marking operation is required to discover and eliminate inter-area cycles; I specifically intend to use indirect marking [Piq95], which uses the IRC tree to perform inter-area marking.

By maintaining separate reference-counts for inter-area pointers between different subsets of areas, we can perform independent inter-area marking on groups of areas to eliminate cycles that lie entirely within those subsets; the practical application I propose is to allow marking at either the node or system-wide levels.

System-wide marking has the distinct problem that it traverses every single link in the heap, which can be a serious problem if the heap significantly exceeds the size of physical memory; deactivating idle areas tries to alleviate this problem. We add an additional reference counter

[1] If B contains a copy of the pointer when the first message is received from A, it immediately send the second message. With the exception of this transient condition, no area is ever the child of more than one parent in the copy-tree, which is why I feel reasonably justified in calling it a tree rather than a digraph.

which is the group of all areas, and use the system-wide reference counters to represent the group of active areas.

An area is deactivated if it is being paged in only when a global marking operation is performed – i.e. it is not in use by user threads. When an area is deactivated, for each entry in its OUT set, it sends a message to the parent area in the corresponding copy-tree which causes the receiving area to decrement the active-system-wide reference count in the parent's entry. Any object that has an active-system-wide reference count which is smaller than its global reference-count is used as part of the root set for marking operations. Thus, deactivated areas need not be paged in to perform a global marking operation. Unfortunately, this does reduce the power of the global marking; it will only reclaim garbage cycles which are entirely contained in active areas. However, I expect it to be a rare event for an inter-node garbage cycle to survive long enough for any containing areas to be deactivated. My simulations will provide some insight as to the accuracy of this expectation.

1.3 Additional design elements

Although not as central to the research aspect of this thesis, the complete memory management system includes a number of additional elements. I will describe each of these aspects in significant detail in the proposed thesis, but it is unlikely that I will actually implement them in the simulation.

Chief among these elements is inter-area object migration, which will dynamically improve locality of reference; although I have some simple heuristic ideas, properly studying this matter is a doctoral thesis in itself. Another element of the overall system is that of heap persistence, and in particular, quick and reliable crash recovery; my scheme for this is based upon entirely straightforward applications of log files and the two-phase commit protocol, and thus is fairly free of research value. A third, cross-cutting aspect is that of concurrent garbage-collection, which would allow user-threads to continue running during (most of) a GC operation; this importance of this issue will be defined by the amount of wasted potential for computation that is measured in the simulated garbage-collection operations.

In this proposal, migration and persistence mechanisms are described in some detail in dedicated chapters. Some mechanisms for concurrent GC are addressed sporadically in the chapter on basic memory management.

1.4 Research contributions

The novel elements of this thesis arise largely from the novel target architecture. In particular, the target architecture is a *parallel* machine, not a distributed one, and thus certain basic assumptions in distributed memory management designs do not necessarily obtain.

To be more specific, one significant contribution of this thesis will be the description and implementation of memory management of an address space which is striped across multiple processors running independent threads of control. Virtually all work on distributed memory management systems has assumed that each object resides entirely upon a single node, while virtually all work on striped memory management has been in the context of SIMD (single shared thread of control) architectures.

Another important contribution of this thesis will be the description, simulation, and evaluation of a few simple hardware mechanisms to perform memory-management bookkeeping in parallel with normal memory accesses. Virtually all work on distributed memory management has assumed that the latency of inter-node memory accesses is so great that bookkeeping operations can be

performed serially without significantly impacting their overall latency; our target architecture, however, promises to offer inter-node latencies which are low enough that such bookkeeping must generally be performed in parallel in order not to have a significant performance impact.

Of equal import, but at a lower level of detail, this thesis will describe the numerous details needed to insure that the target architecture's simple, uncached shared-memory model is preserved even in the face of garbage collection and inter-node object migration; prior works on distributed memory management have neglected these issues almost entirely.

Of course, the specific implementations of indirect reference counting and indirect mark/sweep garbage collection will be new. Three particular aspects stand out as novel: the static maintenance of bookkeeping for different groups of areas which can be independently garbage collected; the dynamic distinction between the management of active and inactive areas; and the delayed applications of IRC updates through both hardware caching and logging to disk.

Finally, one of the gaping holes in almost all of the work to date on distributed memory management systems is the distinct absence of empirical data demonstrating how memory is actually used in these systems. I will provide a wealth of detail as to how several sample applications actually utilize memory on our target architecture, with attention to such details as the frequency of intra- and inter-node inter-area cycles, the actual amount of bookkeeping overhead of my garbage collection scheme, and so on. This type of empirical data is woefully absent from the current literature.

1.5 Proposal organization

The remainder of this document is organized as follows:

Chapter 2 presents an overview of previous work related to this thesis.

The next two chapters describe aspects of the target system and workloads. Chapter 3 describes some key features of the target parallel architecture including memory model, capability and object formats, address-to-node mappings, and my simulation strategy. Chapter 4 describes the test workloads with which the simulated heap will be used, including the parallel language in which they will be written.

The next three chapters describe the heap in its entirety. Chapter 5 describes my scheme for heap memory management, including allocation, areas, distributed hierarchical reference counting, area deactivation, and a host of details necessary to make each element actually work efficiently. Chapter 6 discusses schemes for automatic migration of objects for better locality. Chapter 7 describes my scheme for maintaining heap persistence across node crashes and other failures.

Chapter 8 presents a few of the metrics by which I shall evaluate my memory management scheme. Finally, Chapter 9 concludes with details as to which design elements I plan to implement and evaluate, and a tentative schedule for completing the proposed work.

Chapter 2

Related Work

In this section, I will briefly discuss some related work on unusual addressing mechanisms; on parallel programming languages; and on garbage collecting large, area-based heaps. It is not my intent either in this proposal nor in the final thesis to replicate the efforts of many fine survey works, but rather to present the salient details of those schemes which are directly relevant, either by similarity or by significant contrast, to the approach I am taking.

2.1 Addressing

2.1.1 Referencing objects in very large heaps

In this thesis, my approach to addressing objects in a large virtual address space is simply to reference each object with a capability containing both a large address and object bounds information; a portion of the address identifies which node in a multinode system is responsible for the word at that address.

The Opal operating system described in [CLT92] adheres to the theory that with 64-bit addresses, object references should be simple, direct addresses in a single, shared virtual address space. Opal is targeted at conventional hardware and requires language and compiler support for the storage management system to function properly. My thesis proposes to embed all the information necessary for storage management into hardware-enforced capabilities, enabling arbitrary user code to run without imperiling the system's overall memory-management correctness or safety.

In contrast, the 1989 paper [Mos89] argues that objects should be referenced using short object identifiers which are contextually interpreted. Many of the claims supporting this argument in 1989 have been invalidated by the appearance of 64-bit architectures whose hardware and software deal in 64-bit addresses by default, and for which dealing in smaller word sizes is not generally advantageous. A few of the supporting claims still hold today: a single flat address space discourages heterogeneous hardware, limits the ability of a node to manage its objects independently of other nodes, and by lack of structure, makes it difficult to perform memory-management at an area-based level. My thesis ducks the heterogeneity issue by targeting only homogeneous hardware. The capabilities I use provide the structure required for area-based GC.

Pointer swizzling ([WK92], [Mos92]) is a technique for mapping large heaps into small virtual address spaces. Objects are stored with opaque (indirect) object identifiers (OIDs) which are translated into virtual memory addresses (swizzled) when they are brought into main memory. The two potential advantages of pointer swizzling are first that a large object ID space can be dynamically mapped into a small virtual address space; and second that objects can be arbitrarily

relocated when there are no swizzled pointers to them.

My thesis targets hardware with a large enough address space that the first advantage is not important; however, the second advantage is very important in some cases, and thus the method of deactivating idle areas includes a mechanism which can be viewed as a form of pointer swizzling.

2.1.2 Capability-based addressing

Using capabilities for addressing was initially described in [Fab74]. [Lev84] describes several interesting capability-based architectures. In general, capabilities in these architectures indirect through a central table containing explicit base and bounds information.

[CKD94] describes guarded pointers, a form of capability which includes both an object's actual address and a segment length L which defines how many of the low bits of the address are mutable by pointer arithmetic; the remaining bits of the address are fixed. Thus, the object is defined to be of size and alignment 2^L. This scheme requires a small amount of hardware support to operate. The capability scheme I adopt in this thesis is directly descended from these guarded pointers.

2.2 Parallel languages

There are dozens of programming languages sporting constructs designed to assist with programming parallel systems. They fall into a few, fairly distinct categories, which I will cover here only very briefly.

MIMD languages such as Java [GJS96] and Cilk [Joe96] express parallelism via multithreading and guard shared-visibility side-effects with locks and semaphores.

SIMD languages such as HPFortran [KLS+94], *Lisp [*LI86], and APL [GR74] provide data-parallel primitives which are a thin veneer on standard vector-parallel operations. HPF is notable for providing the programmer with an abundance – perhaps an overabundance – of options controlling the physical distribution and alignment of arrays on distributed-memory multiprocessor systems.

"Idealized" SIMD languages such as Paralation Lisp [Sab88] and NESL[Ble93] express parallelism with parallel-apply operations performed over data collections (typically vectors); nested parallelism is allowed and does not prevent efficient compilation to non-nested, vector-parallel operations. Side-effects that would be visible to other, parallel operations are impossible in NESL, which is purely functional, and forbidden in Paralation Lisp. Paralation Lisp includes some unusual operations, including a mapping mechanism which uses an arbitrary combination function to merge values mapped to the same destination in a binary fan-in tree.

Finally, pH [NAH+95], a parallel dialect of Haskell, discovers parallelism through dataflow analysis. Shared-visibility side-effects are moderated through two special data structures: I-structures, which are write-once slots, and M-structures, which require alternating writes and reads. Reads to empty I- or M-structures block until a value is written.

The parallel version of Scheme I propose in this document draws heavily upon the idealized SIMD languages, but adds mechanisms for communications between parallel operations via controlled side-effects, and, as a "back door", provides the usual MIMD locking and synchronization primitives as well.

2.3 Area-based garbage collection

There are an incredible number of heap designs which are related in some way to this proposal; some are persistent, some are distributed, most are segmented in some way, etc. In this section I

will focus on garbage-collection schemes with particularly compelling relationships or contrasts to this proposal; the final version of the thesis may contain discussion of additional systems.

In particular, in this section I review schemes for garbage-collecting large heaps which rely on breaking the heap up into relatively small, individually-GCable regions; I shall refer to such regions as "areas" after [Bis77], both because Bishop was the first to describe region-based GC and because other terms in the literature such as "partition" or "segment" have preexisting definitions in computer-related contexts.

In general, an area can be independently garbage collected as long as it is known which objects in the area are reachable from other areas; those objects are used as part of the root set for the local garbage collection.

In discussing these systems, certain points of comparison recur:

- How does the system track inter-area object references?
- How does the system eliminate inter-area garbage cycles?
- How does the system terminate a phase of inter-area garbage collection?
- Are object references direct (i.e. pointers) or indirect (opaque)?
- Does the system rely on custom hardware?
- How does the system deal with paging?
- How does the system deal with object migration?

Note that many of the schemes for distributed GC include a wealth of complexity designed to cope with individual node failures, lost, delayed, or duplicated inter-node messages, and so forth; my discussion gives short shrift to these concerns, since I am targeting a tightly-coupled system in which the system is not expected to continue functioning in the presence of node failures, and in which the communications subsystem is expected to deliver messages exactly once, in-order.

2.3.1 GC survey works

Classical GC

Rather than attempt a survey of approaches to uniprocessor garbage collection, I shall refer the reader to Jones' and Lins' *Garbage Collection* [JL96], an excellent survey of traditional garbage collection techniques including, but certainly not limited to, copying garbage collection, mark-sweep GC, and reference-counting. Its coverage of distributed and parallel garbage collection is somewhat sparse.

Distributed GC

Although I shall discuss most of the major approaches to the distributed GC problem below, additional approaches and references are presented in the survey paper [PS95]. Another presentation and detailed analysis of several distributed GC schemes is presented in chapter 2 of the thesis [Mah93]. Additional references may be found in the bibliography paper [San84], which includes not only *avant-garde* topics such as parallel and distributed GC, but also a great many references on classical GC techniques.

2.3.2 Copying GC

System-wide copying GC doesn't actually use areas to any great effect; each processor is responsible for GCing its own region of memory, but GC is system-wide and requires cooperation between all the processors. When a GC is started, each processor begins a copying collection from its local roots. When it encounters a local pointer, it copies as normal; when it encounters a remote pointer, it sends a request to the processor owning the target object which causes that processor to copy from that pointer. This approach is used by the Scheme81 processor [BGH+82] in multiprocessor configurations, and by the distributed garbage collector for the parallel language Id described in [FN96]. Termination detection is an important issue in copying GC. The Id garbage collector detects termination by a messaging protocol,very similar to the scheme described in [Ran83], based on arranging all the processors in a logical ring.

In general, since copying GC requires GCing the entire heap at once, it is inappropriate as the only means of GC in a large parallel system, and is always inappropriate when the size of the heap exceeds that of main memory.

2.3.3 Reference listing

Schemes which use reference listing keep track of which remote areas have pointers to an object in a particular area; the IN list serves as a set of roots for performing independent, area-local GCs which do not traverse references that point out of the area.

Note that reference listing requires that whenever a pointer into area A is copied from an area B to an area C, a message must also be sent to area A to create the appropriate IN entry.

Some reference listing schemes have an entry for every instance of a pointer in a given area; others have one entry for an area regardless how many copies of the pointer there are in that area.

ORSLA

The first area-based garbage collection system in the literature is the reference-listing scheme described in [Bis77] as part of the custom-hardware capability system ORSLA. ORSLA has an IN and OUT entry (actually the same object, an Inter-Area-Link (IAL), threaded onto one area's IN list and the other area's OUT list) for every single inter-area pointer. Whenever an inter-area pointer is created, ORSLA's hardware detects the event and updates the appropriate IN and OUT lists. Area-local GC is performed with a copying collector which copies data objects and IALs; at the end of an area-local GC, the old set of data and IALs is freed.

Inter-area pointers are actually indirected through their IAL under normal circumstances; an area may be "cabled" to another, however, in which case pointers from the first to the second may be "snapped" to point at their targets directly. Note that this means that while the first area may be GCed independently, the second area may only be GCed simultaneously with the first, since its IN list does not record incoming pointers from that area.

ORSLA eliminates garbage cycles by migrating objects which aren't reachable from area-local roots into areas that reach them; in theory, this eventually causes a garbage cycle to collapse to a single area, at which point area-local GC destroys it. [Bis77] doesn't work out details insuring that migration terminates.

For our architecture, the overhead of an IAL for every inter-area pointer would be unacceptable due to the limited per-node physical memory; additionally, having one IAL object serve as both an IN and an OUT entry requires that area-local GC operations require inter-area update messages.

Thor

A reference-listing scheme was designed for Thor [LDS92], a persistent, distributed, object-based database for which garbage collection has been the subject of much research. Thor runs on a heterogeneous collection of conventional hardware; object references are opaque (indirect), thus allowing intra-area object relocation without requiring inter-area communications.

The garbage collection scheme described in [Mah93] uses conservative reference lists between operating regions (ORs) of the database itself, and between ORs and front-end (FE) clients. The scheme is designed to be fault-tolerant in the face of lost or delayed messages.

Each OR maintains a table of objects reachable from other ORs, listing for each object which other ORs might have references. Each OR maintains a similar table for objects reachable from FEs. When an FE or an OR performs a local GC, it sends a "trim" message to each OR listing only those objects still reachable from the OR/FE, thus allowing the OR to clear some entries in the appropriate table. If a trim message is lost, no harm is done, since the table is conservative; the next trim message will fix the problem. Timestamping prevents delayed trim messages from removing entries created after the message was initially sent. Since FE clients are relatively unreliable, FE entries are "leased" – if not renewed periodically, they expire and can be cleared. Note that inter-area garbage cycles aren't reclaimed at all by this scheme.

The problem of inter-area garbage cycles in Thor is taken up in [ML95], which adds a migration scheme on top of the reference listing scheme. Each object is marked with its distance from its nearest root at local GC-time; objects which are only rooted via IN entries start with the distance at that entry and increase it as they proceed. Distances are exchanged with "trim" messages. Thus, while rooted (i.e. reachable) data distances remain finite, the distances associated with garbage cycles will always increase with each round of local GC and trim-message exchanges. A cutoff threshold on distance dictates when migration should occur; an estimated target-node for migration is propagated with distance values, thus causing most objects in a garbage cycle to immediately migrate to the same area.

For our architecture, the overhead of opaque references is too high. More importantly, repeatedly broadcasting trim lists means that bandwidth is *continually* consumed in proportion to the number of inter-area references. Finally, migrating garbage to discover cycles consumes bandwidth in proportion to the amount of garbage, but garbage is the last thing to which we wish to dedicate precious communications bandwidth.

2.3.4 Reference flagging

Reference flagging systems are extremely conservative: each area maintains an IN list which records all objects for which remote references might exist. An entry is created for an object when a reference to that object is first stored into a remote area. The primary advantages of reference flagging are first, that no inter-area messages are needed when a pointer into area A is copied from area B to area C; and second, that IN lists may be extremely compact. The disadvantage is that some form of global knowledge is needed to remove conservative IN entries and inter-area garbage cycles.

Hughes [Hug85] attacks the problem of global garbage collection using a timestamp-based scheme in which inter-area messages are assumed to be reliable, and nodes are assumed to have synchronized clocks. Remote pointers are indirect. Area roots (including IN entries) and remote pointers carry timestamps. Area-local GC propagates the most recent timestamp from roots to remote pointers;- active process objects, the true roots of the system, are timestamped with the time at which the local GC begins.

At the end of the local GC, the timestamps on remote pointers are propagated to the IN entries in their target areas. Each area keeps track of the oldest timestamp that it may not have propagated yet, called the "redo" time. The globally oldest redo time is called "minredo"; at any time, any IN entry which is timestamped with a time older than minredo is garbage and may be deleted. Minredo is calculated using the ring-based termination-detection protocol described in [Ran83].

The reference flagging scheme described in [LL92] uses Hughes' algorithm adapted to cope with unreliable hardware, network, and messages, and also loosely synchronized clocks. Heap segmentation is at the node granularity; intra-node pointers are direct, but inter-node pointers are indirect in order to allow node-local relocation of objects. The scheme relies on a high-reliability central service to preserve certain information across individual node-crashes. Areas occasionally send their OUT lists to the central service; the service uses the collected OUT lists of all areas to compute less-conservative IN lists for each area. Additionally, the Hughes algorithm variant runs entirely on the central service. A variety of additional complexity underlies the correct operation of this system in the face of various message and node failures.

For our architecture, the bandwidth consumed in eliminating conservativism in reference flagging IN lists is simply too great.

2.3.5 Distributed reference counting

In distributed reference counting (DRC) [LM86], for each remotely reachable object in an area, the area maintains a count of the number of remote areas which can reach that object. When an object's reference count goes to zero, it is no longer remotely reachable. DRC has roughly the same messaging requirements as reference listing, but obviously has much smaller memory overhead – one counter per remotely reachable object, regardless of how many areas it is reachable from. Distributed reference counting is much more vulnerable to message loss, duplication, and reordering than reference listing; significant care must be taken to avoid race-conditions. DRC does not collect inter-area garbage cycles.

The DRC-based scheme proposed in [Ng96] for use with the Thor [LDS92] distributed database system is particularly interesting because it uses logging to defer updating reference counts immediately, and thus avoids the need to page in IN entries for an area every time the number of outstanding references changes.

A more complex use of logging for GC Thor is described in [ML96]. In this scheme, each area maintains IN and OUT lists, where an IN list contains precise counts of external references to objects in the partition, while an OUT list precisely identifies all outgoing pointers. Partition-local garbage collection uses objects in the IN list as (part of) the partition's root set. Rather than maintain IN and OUT lists eagerly, this scheme records all object modifications in a globally shared log. The log is scanned to generate an up-to-date IN list prior to garbage collecting a partition. IN and OUT lists are broken into several parts, however, in order to avoid having to go to disk on every update. A number of synthetic benchmarks demonstrate the effects of tuning various parameters related to the sizes of the "basic", "potential", and "delta" lists.

Inter-area cyclic garbage is collected using an incremental marking scheme piggybacked on partition-local garbage collection. Marking begins at global roots; marks are propagated intra-area by local GC, which also pushes marks across OUT pointers. A mark phase terminates when every area has performed a local GC without marking any new data; at that point, any unmarked data may be discarded.

2.3.6 Weighted reference counting

Weighted reference counting was independently described at the same time in both [Bev87] and [WW87]. In weighted reference counting each object has a weight assigned to it at allocation time, and each pointer to that object also has a weight. Pointer weights are powers of two.

When a pointer is duplicated, its weight is reduced by half, and the duplicate receives the other half of the weight; thus, duplicating a pointer requires no communication with the area holding the target object. When a pointer is deleted, a message is sent which causes the weight of the pointer's target object to be decremented by the weight of the pointer. When the object's weight reaches zero, there are no outstanding pointers to the object and it may be reclaimed. There are no synchronization issues with weighted reference counting. WRC does not collect garbage cycles.

One problem with WRC as described is that when a pointer's weight hits one, evenly splitting it becomes impossible. [Bev87] suggests that at this point a new indirection object is created with a large weight; the indirection object contains the weight-one pointer to the actual target object, but the results of the pointer duplication are pointers to the indirection object. This avoids any need to send messages to the original object upon pointer creation, at the cost of memory and indirection overhead.

Note that this scheme as described does not really make use of areas, but it is a a straightforward extension to move the weights to IN and OUT list entries on a per-area basis, rather than maintaining them in every single object and pointer. Such a scheme would reduce the space overhead of normal objects and pointers, and enable arbitrary per-area garbage-collection that would recover intra-area cycles.

[CVvdG90] proposes a slightly different approach to duplicating a pointer whose weight has dropped to one; the weight in a pointer is tagged as being either original or borrowed weight. When a pointer with weight one (original or borrowed) is duplicated, its weight is replaced by a borrowed-weight of MAX, and MAX is added to an entry for the pointer in a *reference weight table*. When a pointer with borrowed weight is destroyed, the borrowed weight is subtracted from the table entry, whereas when a pointer with original weight is destroyed, the weight is subtracted from the original object's weight. The advantage of this scheme is that it avoids loading pointers with indirections; the space overhead is claimed to be roughly equivalent to that of maintaining indirection objects. [CVvdG90] does not address details such as maintaining separate reference weight tables tables on different nodes.

Generational reference counting, proposed in [Gol89], is very similar to weighted reference counting; instead of maintaining an individual weight, however, an object keeps a ledger counting outstanding pointers of each of several generations, and a pointer keeps track of its own generation, and the number of children that have been copied from it. When a pointer is destroyed, a message is sent to its target object which decrements the ledger entry for its generation, but increments the entry for the next generation by the size of its children-count.

For our architecture, the small communications requirement of WRC is very appealing, but the need to build up indirection chains is unacceptable.

2.3.7 Indirect reference counting

The problem with weighted reference counting is what to do when one runs out of weight to divide amongst pointer copies. Indirect reference counting [Piq91] (IRC) avoids the problem by counting up, instead of down. Specifically, when a pointer P is copied from area A to area B, area A increments a reference count associated with P, and area B records the fact that the pointer came from area A. When all copies of P are finally eliminated from B, if B's reference count for P is zero,

B sends a message to A decrementing A's reference count for P. Thus, the total reference count of P is represented in a tree whose structure mimics the inter-area diffusion pattern of P; the root of the tree is the area containing P's target, and children point to their parents, rather than the other way around. Note that as long as B still has a copy of P, even if A eliminates all of its copies, it must preserve the reference count entry for P as long as it is nonzero. Area-local GC operates using any local object with a nonzero reference count as a root.

The communications overhead of IRC is one extra message per inter-area pointer copy – the decrement message sent when an area discovers it no longer has copies of a pointer it has received. The space overhead is the record at each intermediate node in the diffusion tree which notes the local outstanding reference count, and from which area the pointer was originally received.

Object migration is relatively easy with IRC – the object is migrated to its new location, and the root of the IRC tree is made an internal node, with the new root as its parent.

In the worst case, a single remote area holding a copy of P may lock in place an arbitrarily long chain of records through other remote areas. [Mor98] presents a scheme similar to IRC, but which eliminates these chains via diffusion-tree reorganization. In this scheme, when an area B is to send a pointer P to area C, where P points into A, the following sequence happens: B increments its local reference count for P; B sends the pointer to C; C sends an "increment-decrement" message to A; A increments its local reference count for P; A sends a decrement message to B; B decrements its local reference count for P. Thus, at the end of the flurry of messages, all diffusion tree leaves have area A as their parent – and since area A is where P's target object lives, they need not explicitly record their parent area.

This scheme thus has two advantages: first, it avoids long chains of records through areas that otherwise have no copies of a pointer; second, records may be smaller since they don't need to record a parent area distinct from the pointer's target area. The disadvantage, however, is that the communications overhead for a pointer copy is now three messages instead of one: the "increment-decrement" message to the home area, the "decrement" message to the sending area, and then eventually the "decrement" message to the home area. Only the last of these is needed in straight IRC.

For our architecture, simple IRC is very appealing due to its low communications requirements and relatively low memory overhead.

2.3.8 Grouped garbage collection

Weighted or indirect reference counting alone does not collect of inter-area garbage cycles. One approach to solving this problem suggested in [LQP92] and [Piq96] is to dynamically form groups of areas, and garbage collect them together. [LQP92] describes how to do such grouping with a mark-sweep algorithm on top of distributed reference counting; [Piq96] extends the grouping scheme to work with indirect reference counting. Groups may be dynamically formed, and can exclude failed nodes in order to reclaim garbage cycles which do not pass through failed areas.

My garbage collection scheme will make use of a simplified version of grouping to allow garbage collection at a few different granularities.

2.3.9 Other indirect GC schemes

As pointed out in [Piq96], nearly any GC algorithm can be modified to be an indirect algorithm by altering it to traverse parent pointers in the IRC tree instead of directly following remote pointers to their targets.

For instance, indirect mark and sweep [Piq95] builds on top of the IRC diffusion tree by propagating marks normally within an area, but when marking from a remote pointer, following the parent link in the remote pointer's copy-tree entry. When the marking phase is completed, any object or copy-tree entry which hasn't been marked may be eliminated in the sweep phase (note that sweeping does need to keep reference counts consistent in surviving portions of IRC trees.)

Another indirect scheme is indirect reference listing (IRL) [PV98], in which counters in an IRC tree are replaced by explicit entries, one for each area to which a pointer has been copied. IRL is somewhat more resistant to individual node failures than IRC, although it is still not resistant to message loss or duplication.

The SSP chains scheme [SDP92] is similar to indirect reference listing, but includes a timestamping protocol which adds tolerance for duplicated, lost, delayed, and reordered messages, as well as individual node crashes. [LFPS98] extends SSP chains with a timestamp propagation scheme inspired by [Hug85] that eventually eliminates inter-area cycles.

Chapter 3

Target System

The eventual target architecture for our heap is the Aries parallel capability machine, a distributed-memory multiprocessor with very fast (i.e. low-latency) interconnect providing a completely uncached non-uniform memory access (NUMA) shared-memory model. In this section I describe the characteristics of the target system which are important to the heap design, while ignoring everything that isn't directly relevant; I also describe my approach to simulating the important elements of the system in an easy-to-program-to framework.

3.1 Processing node

A node in the target system contains a processor tightly coupled with a relatively small (1-2MByte) quantity of volatile main memory. Each node has a dedicated network interface (NI) component to handle internode messaging; the NI is also closely coupled to the main memory. Additionally, each node or small cluster of nodes has a large secondary storage device – several hundred Mbytes of disk – for paging and stable storage purposes. Because each node, including processor, network interface, and main memory, is implemented entirely on a single piece of silicon (in fact, many nodes may be implemented on the same chip) the processor-to-main-memory ratio is fixed.

3.2 Addressing

Addressing is based on a shared-memory model: a single, shared virtual address space encompasses all the storage on all nodes. Threads of control (processes) are protected from each other by the use of capabilities – hardware-recognized pointers with embedded bounds and access-control information – to provide a much finer protection granularity than that provided by conventional architectures.

A given address has three components: a physical node identifier, a virtual page number, and an offset. Thus, to find the value stored at an address $A = [N, VP, O]$, the read request is routed to node N; node N translates the virtual page number VP to a physical page number PP, paging data from secondary storage if needed; and then finally the value stored at offset O in the physical page is returned.

Addressing is performed at single-byte granularity.

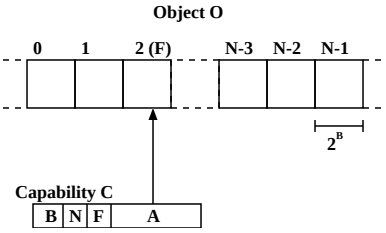

Figure 3.1: The format of a capability C pointing into an object O.

3.3 Capability (pointer) format

In the eventual target architecture, capabilities will include both simple object-bounds information, and sophisticated access control information. For the purposes of memory management, however, the access control elements are irrelevant, and thus I will ignore them entirely from here on.

All pointers manipulated by non-privileged code are actually "capabilities" which specify not only a target address, but also the bounds of the object in which the address lies. The hardware uses these bounds to dynamically detect access violations. Thus, even without more sophisticated access-control, capabilities provide a much finer-grained mechanism for inter-process protection and cooperation than the traditional page-based protections of conventional architectures.

The specific format of a capability C for an address A in object O is $C = [B, N, F, A]$ where O is composed of N blocks of 2^B words each (allocated block-aligned;) and F is the number of the block into which A points. See Figure 3.1. The presence of F allows the rapid discovery of the beginning and end of the object regardless of where the address A is pointing.

3.4 Address-based striping

Since each processor in the target system is running independent threads of control, it would be awkward to have an addressing scheme which always striped objects across multiple nodes. However, since we have a massively parallel system, it would be equally unwieldy to be unable to allocate objects such as vectors which did not span multiple nodes. Thus, our target system allows striping at arbitrary power-of-two granularities, based on information contained in the address.

Specifically, each address A can be viewed in the format $A = [S, D]$. S is a group of bits large enough to hold an index to any position in the remaining bits, D. The index S indicates where in D the physical node ID is embedded. Thus, an address is of the form $[S, V_h, N, V_l]$ where V_h is the high bits of the virtual address; N is the identifier of the node responsible for this particular word of data; and V_l is the S low bits of the virtual address.

For example, an S of 0 indicates that that the physical node ID is composed of the lowest bits of the address. This means that when the address is incremented by one, the physical node ID is incremented by one – this results in an object which is striped across the system's nodes at one byte per node.

An S of 6, on the other hand, places the node ID 6 bits into D, so it changes only with every 64th increment – thus, the object identified by such an address is striped at 64 bytes per node.

The largest S possible, of course, stripes addresses at an incredibly coarse granularity; in practice, rather than actually striping objects at this granularity, this coarse-space is where small, non-striped objects are allocated in order to reside on only one processor, and where node-local

hardware provides memory-mapped interfaces.

3.5 COTS simulation strategy

For purposes of implementing and evaluating a parallel heap, I will write a very high-level simulator of the important features of the target architecture; I will ignore details of the eventual architecture that are unimportant, focusing instead on making the simulator an easy compilation/translation target. The simulator will run on a Compaq Alpha workstation, in single-threaded fashion.

Each simulated processor will be a simple stack processor. A handful of registers will be provided to hold capabilities to important objects such as the stack; the environment; the program code; and the interrupt vector.

Simulated secondary storage devices will be backed by real files on the simulation platform, in order to provide true heap persistence.

The simulated communications network between processing nodes will be very simple as well, accepting and delivering messages with no failures. If convenient, the interface to the messaging system will also be designed for compatibility with a cycle-accurate network simulation currently being written by others in our research group.

Processor, disk, and network simulation components will all collect copious profiling information to enable accurate performance evaluation of the memory management system.

Chapter 4

Test Workload

In this section I describe the workloads I will use to drive the memory management system. In particular, I describe the parallel variant of the Scheme programming language in which I will write the workload applications, and then describe some of the applications themselves.

4.1 Parallel extensions to *Scheme*

My parallel extensions to scheme are designed to abstract away most of the horrific details of parallel programming as thoroughly as possible. This section presents the key constructs and procedures, but elides many obvious/unimportant procedures.

4.1.1 Parrays

The primary data structure for parallelism is the *parray*, an array of arbitrarily large dimensionality whose elements are distributed across the processing nodes. I will refer to an individual element in a parray as being at point P, where P defines the index along each dimension; a point is represented simply as a list of the indices. If we hold some indices fixed, and allow others to vary, the fixed set of indices defines a subspace S; a subspace is represented as a list of pairs, where the first element of each pair is the number of the dimension being fixed, and the second element is the index to which it is fixed.

4.1.2 Parray allocation

A parray's layout over the processing nodes is determined at allocation time. I have tried to keep the complexity well under that of the array layout options in high performance Fortran [KLS+94], but a variety of options still remain. Although this complexity is available, however, I expect that default allocation strategies involving no programmer-specified options will suffice in most circumstances.

Default parray striping strategy

By default, a parray which is larger than the number of the nodes in the system will be allocated in a striping-space with striping granularity such that the array either stripes across all the nodes exactly once, or else "wraps around" the set of nodes once (assuming the parray is large enough to wrap around at all.) This approach insures two properties: every node holds some of the parray's

elements, with no node holding more than twice as much as another; and as many adjacent elements as possible (within the limits imposed by the first property) are co-located on the same node.

By way of example, a one-dimensional parray (AKA a pvector) with 4N elements allocated over N nodes will place four elements per node: elements 0-3 on one node, elements 4-7 on the next node, and so forth. A pvector with 5N elements would be similarly allocated with elements in groups of four; in this case, the first node receives not only elements 0-3, but also, on the wraparound, gets elements 4N through 4N+3. On the other hand, a pvector with 4N-2 elements would be allocated in two-element groups; the first stripe across the nodes would take care of the first 2N elements, with the remaining 2N-2 elements being allocated over all but the last node in the second stripe.

The assumption underlying this default strategy is that operations on a parray will frequently involve parallel-prefix operations such as scan and reduce which benefit from minimizing the physical distance between adjacent elements, and thus grouping adjacent elements on the same processing node is better than distributing them with adjacent elements on different processing node. Allowing one wraparound insures that (for a large parray) no node is left without any of the parray, which would leave the node entirely idle during operations on that parray.

`make-parray`

The typical way of creating a parray is using this procedure:

`(make-parray` *sizes* `['fill` *fillproc*`] ['spread` *dimorder*`] ['align` *align-parray*`])`

Sizes is a required list of the sizes of each dimension of the new parray; in the event that the parray has one dimension, the list may be replaced with an integer. Multidimensional arrays may be padded in some dimensions to fill out to power-of-two sizes in order to map cleanly into the underlying hardware's striping-spaces. Each size argument may either be a simple integer, or a list. In the latter case, the list has this format:

`(`*dimsize* `['cluster` *clustersize*`] ['interleave])`

dimsize specifies the size of the dimension. The two options enable deviation from the default allocation strategy described above.

clustersize: The optional *clustersize* specifies the power-of-two granularity at which elements should be clustered – i.e. the array is distributed over the nodes as if this dimension were of size $dimsize/2^{clustersize}$. *clustersize* is limited to powers-of-two in order to interface cleanly to the underlying hardware, which supports striping at granularities which are powers of two. For a pvector of size 5N, if a *clustersize* of 3 had been specified, then elements 0-7 would be placed on the first node, elements 8-15 on the second, and so forth, thus only striping over a subset of the processors.

interleave: The optional `'interleave` specifies that each consecutive cluster is allocated on the next node in the system, instead locating multiple adjacent clusters on the same node to avoid multiple wraparounds. This approach maximizes the parallel distribution of the array, insuring that no processing node is responsible for more than one more element cluster than any other processing node.

Remaining `make-parray` arguments: Back at the `make-parray` arguments: *fillproc* is an optional procedure which will be invoked once for each element in the parray. *fillproc* receives as an

argument the element's point in the parray; its return value is used to initialize the element. If no *fillproc* is specified, the initial value of the elements in the parray are unspecified.

While a parray may be of arbitrary dimensionality, a real machine may be of at most three dimensions, so picking which dimensions should be parallelized, and which should be clustered onto single nodes, is potentially important. Thus, *dimorder* is an optional list of (lists of) dimension identifiers; the order of the list specifies the order in which dimensions should be striped across the processing nodes. Sublists specify dimensions that should be striped with equal priority, i.e. in "block-major" fashion. For instance, an indices of '(3 (1 2) 0) would specify that the most important dimension to spread out is dimension 3, then dimensions 1 and 2 simultaneously (in blocks), with dimension 0 taking up the rear.

Finally, if the optional *align-parray* argument is specified, the element of the new array at indices I will be located on the same node as the element of align-parray with indices I; performing elementwise operations on aligned arrays is obviously much less communications-intensive than otherwise. *align-parray* must be of identical dimensionality (including clustering, interleaving, spread, etc.) as the new array; if any of the optional dimensional parameters aren't specified for the new array, they are inherited from *align-array*.

4.1.3 Parallel-prefix operations

A great many parallel tasks may be completed using nothing but parrays and the data-parallel operations preduce and pscan.

`(preduce` *op base target-parray [dimension]*`)`
`(pscan` *op base target-parray [dimension]*`)`

In these functions, *op* is a two-input associative function; base is the identity value for *op*; target-parray is the array to be operated on; and the optional dimension specifies along which dimension the operation should be performed (defaulting to dimension 0). If *op* is not an associative function, or base is not the identity value for op, the end result is unspecified.

preduce returns a parray of one less dimension than target-parray (or an integer if target-parray is single-dimensioned.) The entry in the returned parray for element at point P is the result of using *op* to combine all of the elements in the single-dimension subspace in which all dimensions but *dimension* are held fixed at the indices specified by P. Combination is applied only to adjacent elements or partial results, but the order in which such combinations are performed is arbitrary – as long as *op* is truly associative, however, the end-result will be deterministic.

The actual implementation will combine adjacent elements located on the same node serially, by simply applying *op* in order; the partial results computed at each node will then be combined in parallel-prefix fashion to produce the final result.

By way of example,
`(preduce + 0 number-vector)` where number-vector is a pvector of integers will return the sum of all of the numbers in the pvector. Similarly,
`(preduce * 1 pmatrix)` will return a vector whose Ith entry is the product of the elements in the Ith row of pmatrix.

pscan returns a parray of the same dimensionality as target-parray. In this case, the values along dimension in the result are the partial results of applying *op* along that dimension.

By way of example, (pscan + 0 number-vector) will return a vector whose first element is just the first element of number-vector; the second element is the sum of the first and second elements of number-vector; the third element is the sum of the first through third elements of number-vector;

and so on.

pscan and preduce block until all invocations of *op* have completed.

4.1.4 Parallel map

`(pmap` *op parray0* `...)`

op is a procedure taking N+1 arguments, where N is the number of parrays that are passed as arguments. The argument parrays must all have identical basic dimensionality (although clustering, interleave, etc. may differ.) pmap returns a parray of the same dimensionality as the argument parrays, aligned with parray0; the element at point P contains the result of applying *op* to arguments consisting of the elements at P in each of the argument parrays, plus the point P itself. pmap blocks until all invocations of *op* have returned.

Although parallel map looks data-parallel at first glance, it is really control-parallel in that the operation invoked at each element runs as its own thread of control.

4.1.5 Combiners and splitters

Since multiple threads of control can perform side-effects on objects visible to one another, we need mechanisms for performing shared-visibility side-effects in a controlled fashion. As I will discuss further on, I do provide the old standbys of locks and so forth, but I believe that in many cases the use of combiners and splitters will be a much cleaner means of controlling inter-thread side-effect management.

A combiner is a two-input, one-output function used to merge parallel attempts to side-effect the same location. A splitter is a three-input, two-output function used to split the result of the combination operation to provide distinct return values for the various merged calls.

`(combine!` *variable value combiner [splitter]*)

variable names the slot targeted for side-effecting; value is the base value for combination; combiner is a two-input, one-output procedure; and splitter is an optional three-input, two output (one pair output) procedure.

We begin our description by ignoring splitter.

When a thread calls combine! on a variable X with value Vcomb and combiner C, if no other thread calls combine! around the same time, then the value V_{curr} currently in X is read, the value $V_{res} = C(V_{curr}, V_{comb})$ is computed, and then V_{res} is written back into X. This operation is atomic with respect to other combiners attempting to side-effect X, although if C performs additional side-effects to other variables, or if non-combining side-effects are performed on X, no guarantees are made and non-deterministic behavior is expected. Without a splitter provided, the return value of combine! is unspecified.

When two threads call combine! on a variable X with values V_1 and V_2, but with the same combiner C, the system has the option of combining the two calls into one by computing $V_{comb} = C(V_1, V_2)$, then using V_{comb} to perform the combination into the target variable V. Such combination may happen recursively, forming a dynamic parallel-prefix reduction tree. In the event that C is both associative and commutative, the result will be deterministic.

Now that I have described the basic use of combiners, let us add a splitter S to the combine! call. In the case in which only one thread calls the combine! operation, S is not actually invoked, but the combine! call returns the value V_res that was written back into X.

However, in the case in which combine! calls from multiple threads are merged, the dynamically formed reduction tree structure is remembered; when a final value $V_{res} = C(V_{left}t, V_{right})$ is written into X, the pair $(R_{left}, R_{right}) = S(V_{left}, V_{right}, V_{res})$ is computed, with the value R_{left} being returned up the combination-tree to the left, and R_{right} to the right. These values will be recursively split with S at each combination-point in the tree until distinct values are finally returned to each thread invoking combine!

There is a form of combine! for every form of set!:

```
(vector-combine!  ...
(parray-combine!  ...
(combine-car!  ...
(combine-cdr!  ...
```

4.1.6 Primitive synchronization

Occasionally, explicit synchronization mechanisms will be needed. I provide the following primitive mechanisms with minimal description; they are modeled directly on the Java synchronization mechanisms.

`(lock` *obj*`)`

obj must be a distinguished lockable object. The call to lock will block if another thread has already locked obj.

`(unlock` *obj*`)`

This unlocks obj; if there are any threads blocked waiting to lock obj, one of them will now acquire it.

`(wait` *obj*`)`

This may only be called by a thread which has already locked obj; the thread goes to sleep while simultaneously releasing the lock on obj. When the thread is reawakened by a notify or notify-all call, it will reacquire the lock before continuing.

`(notify` *obj*`)`

This may only be called by a thread which has already locked obj. It causes exactly one thread which is waiting on obj to wake up; the newly-awakened thread will not make forward progress until the lock is released so that it can reacquire it.

`(notify-all` *obj*`)`

This may only be called by a thread which has already locked obj. It causes all threads waiting on obj to wake up.

`(synchronized` *obj body*`)`

This is syntactic sugar which locks obj, executes body, and then unlocks obj.

4.1.7 Primitive control flow

Although I hope that in most cases pscan, preduce, and pmap will prove adequate means of launching parallel operations, in this section I discuss two primitives for manipulating control-flow more explicitly. These primitives will prove important in implementing the higher-level constructs, as well.

`(on-proc-apply` *procid func arglist*`)`

This procedure mirrors the functionality of scheme's apply procedure, with the additional property that it explicitly specifies upon which processor the function func should be invoked. This function might be used when traversing a data structure spanning multiple nodes, for instance.

`(fork-proc-apply` *procid pid func arglist*`)`

This procedure returns a thread object; a thread is started on the target processor in which the procedure func is applied to the arguments in arglist. The thread object may be queried as to the thread's status, and when func finally returns, the thread object can be queried for func's return value.

4.1.8 Capture

By default, Scheme passes objects by reference, rather than by value. Although efficient on a uniform memory-access time platform, on a non-uniform memory-access machine, it can have unfortunate performance implications, as a procedure P running on processor B repeatedly accesses objects on the processor A from which P was spawned.

In order to help alleviate this problem, the capture function may be wrapped around arguments to a function call, and around arguments in the arglist for apply, apply-on-proc, fork-on-proc, etc.

The behavior of capture varies by the type of the captured value. Numbers, lists, and vectors are explicitly deep-copied, while only the reference to a parray is copied – the parray itself is not duplicated. User-created data types may define their own behavior when captured.

The crucially important element of copies made by capture is that the copied data structures are allocated on the processor upon which the function being invoked will execute; thus, references within the function to its arguments will not require references to be made across the network to other nodes.

4.1.9 Capturing-lambda

In Scheme, the body of a lambda expression may refer to variables defined in an environment outside of the lambda itself. When a lambda is defined on processor A, however, but invoked on B, these bindings can be a source of inefficiency, as all references to them must travel over the network to A.

The special form capturing-lambda attempts to address this problem.

`(capturing-lambda` *(arg0 arg1 ...) (body... unbound-var0 .. unbound-var1...)*`)`

has the property that when the c/lambda is actually applied, the unbound variables in the body are bound to the values returned by application of the "capture" function to the variables they name in the enclosing environment; thus, the executing body need never again refer to the bindings of variables in the defining environment.

Since the behavior of the "capture" function on most values is to simply copy the target value, when a c/lambda has a reference to an unbound variable X in it, after the c/lambda has been applied, the value of X as seen from within the c/lambda is fixed at whatever it was in the environment in which the c/lambda was created. (I.e. capturing-lambda has an implicit call-by-value effect on most of its arguments, both explicit and implicit.)

(c/lambda's parameter-passing behavior is not unlike the Java RMI's, except that RMI doesn't have a different syntax from normal function calls, so you can confuse yourself as to what's getting copied and what's getting passed by reference when you're mixing RMI and normal calls.)

One final semantic note: the binding happens when a c/lambda is actually applied. So if you repeatedly apply a c/lambda, new bindings will be made for each application (instantiation.)

The definition of c/lambda is likely to evolve rapidly under actual usage conditions.

4.2 Benchmark applications

In this section I briefly present the test workload applications I will use to generate memory management demands on our system. I will have four major applications: a rapidly changing spatial index; a fast fourier transform; a symbolic AI application; and a spatial simulation.

4.2.1 Database search application: R-tree spatial index

An R-tree is a height-balanced, potentially sparse tree whose leaves are data objects keyed with a multidimensional bounding box, and in which each internal node is marked with the minimal bounding box containing all the objects under it; these bounding boxes may overlap.

The three operations on an R-tree are insert, delete, and query. Insert attempts to place a new data object into the tree in a fashion that minimizes the overlap of the bounding boxes along different paths. Query searches for all objects whose bounding-boxes intersect a key bounding-box; the query operation may have to search along many paths in the tree to find all matching objects. Delete is like query, but removes all objects that match the search key from the index.

As part of the benchmark suite for the Data Intensive Systems DARPA program our group is operating under, we have received an R-tree index implemented in C, as well as a number of test data sets. I will recode this R-tree in parallel scheme, and use the unmodified test data sets.

4.2.2 Numerically intensive application: FFT

Another benchmark from the Data Intensive Systems program we are operating under is a fast fourier transform (FFT) benchmark. Since the FFT routine employed by the DIS benchmark is both extremely complex, and written in C, I will write a simple FFT routine in parallel scheme; I will, however, use the DIS test data sets to drive it.

4.2.3 Symbolic AI application: ?

I am currently looking for a suitable symbolic AI application; I am particularly interested in an application which builds up densely connected data structures which are amenable to parallel traversal and modification.

4.2.4 Physical (spatial) simulation: ?

I am currently looking for a good physical spatial simulation application. Under current consideration are N-body problem solvers and weather simulations, but other options are possible.

Chapter 5

Proposed Memory Management Strategy

In this chapter, I finally address the issue of performing memory management on the system described in Section 3. I present approaches for solving different parts of the problem, pulling them together gradually. I should note here that this section represents a snapshot of several rapidly changing designs; specifics and even generalities may be subject to sweeping change before the final version of this thesis.

All memory management code will run in privileged mode, so it will be able to break the rules enforced on user code by capabilities.

The first several sections of this chapter describe the memory management scheme in terms of non-striped objects; the last section explains how to extend the scheme to include striped objects. Specifically:

Section 5.1 describes the system's object format – this format obtains for both non-striped and striped objects. Section 5.2 defines areas and threads. Section 5.3 describes my proposed implementation of indirect reference counting. Section 5.4 describes my implementation of indirect marking, and my variant of grouping areas to allow marking at the node-level as well as the system-level. Section 5.5 explains how forwarding pointers are used to enable object relocation. Section 5.6 explains how areas are deactivated, thus removing themselves from system-wide garbage collection efforts. Section 5.7 describes a few heuristics for deciding when to apply each type of garbage collection. Finally, Section 5.8 explains how to extend the schemes already described to allocate and garbage-collect striped objects.

5.1 Object format

An object has a very simple format: it is an address-contiguous group of words of data. An object which is large enough that the capability pointing to it has blocksize greater than one may have to be padded with a few extra words; in that case, each padding word has an "interrupt on access" bit set, and contains the actual length of the object.

This object representation allows hardware to always catch an access past the intended end of an object, even when the object had to be padded due to the limited resolution of the capability bounds format.

5.2 Areas and threads

5.2.1 Areas

The heap is partitioned into *areas* which are distributed over the processing nodes; the term "area" is adopted from [Bis77]. An area is the basic unit of garbage collection in this GC scheme. An area resides entirely on one node, but a node is responsible for many areas.

An *area* is a collection of segments of address-space, all of which reside entirely on one processing node. Although an area encompasses a huge number of virtual addresses, it only has a few pages of content, as more than one area should be able to fit in a node's physical memory at one time.

The precise explanation of which segments of address space are bundled together in a single area is complex, due to the need to handle striped objects in a sensible fashion; I thus defer the precise definition until Section 5.8, in which I discuss how to manage striped objects. For now, an area should be thought of as a simple, contiguous chunk of address space.

I refer to the area into which a pointer points as that pointer's "home" area; other areas are "remote" with respect to that pointer.

An area is usually garbage-collected independently; pointers out of the area are not normally traversed. Each area has an IN set listing objects reachable from remote areas, all of which are used as (part of) the root set for the area-local garbage collection. (The mechanics of maintaining the IN set are explained in the section on indirect reference counting, Section 5.3.) My implementation of area-local GC will be a very simple stop-and-copy garbage collection.

5.2.2 Threads

A thread is a single stream of control. A thread's state conceptually lives in a specific area. A thread allocates non-striped objects within its own area (unless the area fills.) References made by a thread into its home area are "local", and references into other areas are "remote".

We can keep the "operating area" definition in a thread-specific register, thus making it very easy to discover if a particular pointer lies inside the current area or not.

5.2.3 Simple object allocation

At this point, we are ready to describe the extremely simple process by which node-local objects are allocated.

When a thread needs to allocate a new, non-striped object, it calls a system routine to perform the allocation. For most allocations, the system routine simply advances a counter which indicates the end of the currently-allocated region of memory, and returns a pointer to the newly allocated memory. This is only complicated when there isn't enough storage space currently allocated to back the area's address-space to handle the new allocation.

If varying degrees of garbage collection (see Section 5.7) fail to reclaim enough space to honor the request, the allocation routine must perform one of two actions: either it must allocate additional backing storage to the current area, or, if doing so would make the area too large, it must allocate the object in a different area. A sufficiently large object may, in fact, merit a single area all to itself! The exact parameters guiding these decisions are a matter for engineering experimentation.

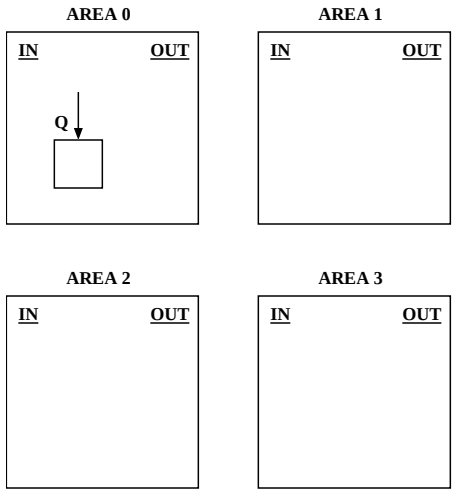

Figure 5.1: Object Q allocated in Area 0.

5.3 Indirect reference counting for inter-area GC

In this section, I describe my implementation of *indirect reference counting* (IRC), a scheme that counts the number of areas containing a pointer, rather than counting individual copies of the pointer within areas. Piquer first described IRC in [Piq91]; in this section I discuss both the general idea of IRC, and my specific approach to implementing it for the target system.

The simple idea behind IRC is this: Each area maintains a reference count for each pointer it has sent to another area, and remembers from which area it received each foreign pointer. This generates a per-pointer copy-tree (diffusion-tree) of reference counts, based on the pattern in which the pointer is initially distributed. As copies of a pointer are destroyed, the leaves of the tree prune themselves and notify their parents, resulting in a collapse of the tree which mirrors its construction both temporally and in terms of communications pattern. Once the reference count at the root hits zero, area-local memory management has complete control over the object.

In the following subsections I describe my approach to IRC in significantly more detail. Sections 5.3.1, 5.3.2, and 5.3.3 cover the basic ideas; Section 5.3.4 discusses exactly how to implement the data structures on our target architecture in an efficient fashion. Finally, Section 5.3.5 explains the design rationale underlying my selection of IRC and this specific IRC implementation. Note that as described in this section, objects must never be relocated by area-local GC; I explain how to eliminate this constraint using forwarding pointers in Section 5.5.

5.3.1 Counting up

First I describe the construction of a hierarchical reference count tree. Consider the example in Figures 5.1-5.3.

In Figure 5.1, an object Q is allocated in Area 0. Note that each area maintains an IN set and an OUT set; entries in the IN set correspond to pointers coming into the area pointing at local objects, while entries in the OUT set correspond to pointers pointing out of this area to remote objects.

In Figure 5.2, a pointer to Q is copied from Area 0 to Area 1, resulting in entries in the OUT set on Area 0 and the IN set on Area 1.

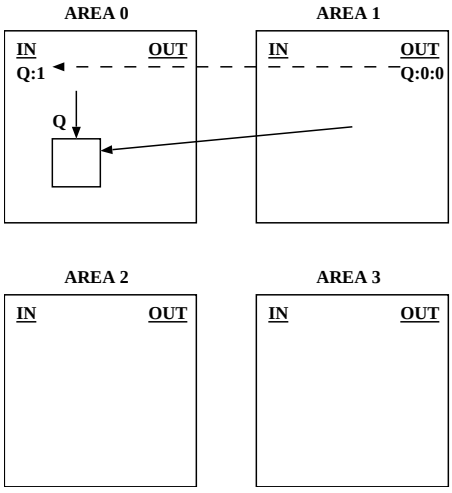

Figure 5.2: Pointer to Q copied from Area 0 to Area 1. The copy-tree links are shown as dashed lines.

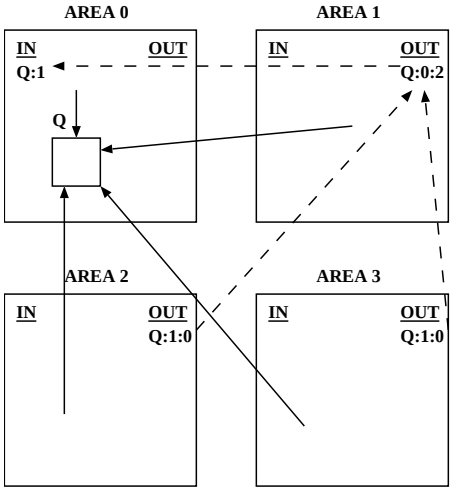

Figure 5.3: Pointer to Q copied from Area 1 to Area 2 and 3.

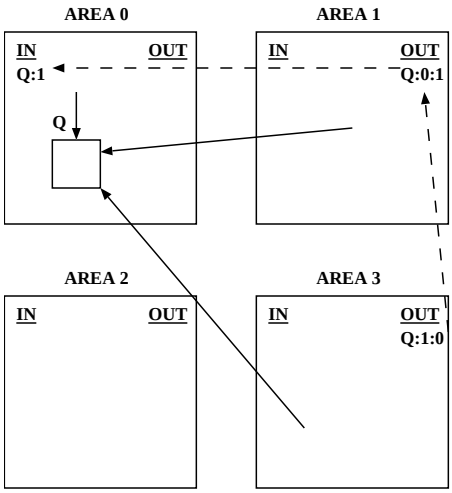

Figure 5.4: All pointers to Q in Area 2 have been destroyed.

The IN list record format is simple: the object identifier is followed by the count of how many times its pointer has been sent to a remote area. The OUT list record is only slightly more complex: the object identifier is followed by the identity of the area from which the pointer was initially received, and then by a reference count of the number of areas to which the pointer has been subsequently sent. In this case, one copy of the pointer has been sent out of Area 0, and no copies of the pointer have been sent out of Area 1, so the IN entry on Area 0 is Q:1, and the OUT entry on Area 1 is Q:0:2.

In Figure 5.3, a pointer to Q is copied from Area 1 to Areas 2 and 3; the reference count in Area 1's OUT entry for Q is incremented twice to record the transfers, and each of Areas 2 and 3 record the fact that they received the pointer from Area 1.

Note that constructing the hierarchical reference count tree requires no communications overhead – its construction results from the transmission of pointers from one area to another. In particular, unlike conventional reference counting schemes, no special communications are needed with Area 0, even though the copied pointer refers to an object stored there!

5.3.2 Counting down

Now I examine how the hierarchical reference count tree just described collapses as local copies of pointers to Q are destroyed.

In Figure 5.4, all pointers to Q in Area 2 have been destroyed; the entry for Q in Area 2's OUT set has been removed, and Area 1's reference count for Q has been decremented. This operation requires one message to be sent from Area 2 to Area 1 containing the decrement order.

Note that the mechanism for discovering that all pointers to Q in Area 2 have been destroyed doesn't have to be instant; as long as the discovery is made eventually, the reference counts will be decremented appropriately. The mechanism itself is unimportant; a area-local garbage collection will do the trick in this implementation, but other schemes could be envisioned as well.

In Figure 5.5, all pointers to Q in Area 1 have been destroyed. When this fact is discovered, the OUT entry for Q is flagged, but since its reference count is non-zero, it is not deleted, and no messages are sent.

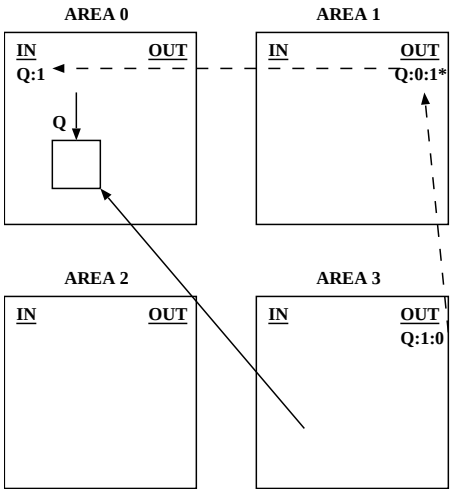

Figure 5.5: All pointers to Q in Area 1 have been destroyed.

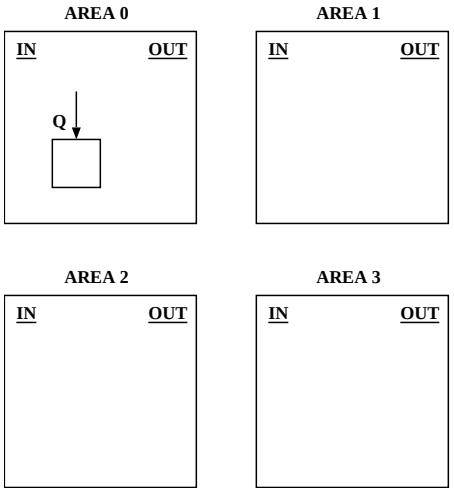

Figure 5.6: All pointers to Q in Area 3 have been destroyed.

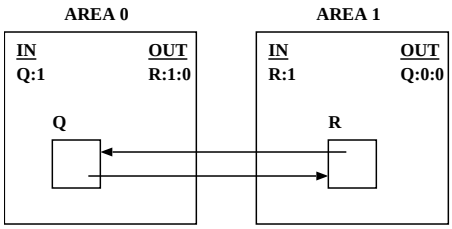

Figure 5.7: Objects Q and R form an inter-area cycle.

In Figure 5.6, all pointers to Q in Area 3 have been destroyed. The decrement message sent from Area 3 to Area 1 allows 1 to finally clear its OUT entry and send a decrement message to Area 0 which can in turn clear its IN entry.

Note that overall, one decrement message is sent for each initial inter-area pointer-copy, between the same two areas in each case.

5.3.3 Corner cases

Receiving the same pointer multiple times. Consider again the hierarchical reference tree in Figure 5.3. The question arises: in the course of continuing computation, what happens if a pointer to Q is copied from, say, Area 2 to Area 3, or even from Area 1 to Area 3 a second time?

The answer is simple: as soon as the copy arrives at Area 3, and Q is looked up in the OUT list and found to already be in Area 3, a decrement message is immediately sent back to the source area; this insures that there is only one path through Q's copy-tree tree from any area in the tree to the root-area where Q actually lives. As always, one decrement message is sent for each inter-area pointer copy – there is simply no delay between the copy and the decrement when the receiving area has previously received a copy of the pointer.

An object with only remote pointers. It is possible for an object to be reachable only from remote pointers. In this case, the fact that remote pointers exist is recorded by the object's entry in the area's IN set. Area-local garbage collection must therefore use the IN set as (part of) the root set.

Inter-area cycles. Obviously, the reference counts on inter-area cycles such as shown in Figure 5.7 will never go to 0, even though neither object is really reachable anymore. An additional garbage collection mechanism, described in Section 5.4 is needed to identify and eliminate these cycles.

5.3.4 IN and OUT set implementation

In order to adjust reference counts efficiently, IN and OUT entries must found quickly; thus, we turn to hashtables. Although the IN and OUT sets are conceptually distinct, we can implement them as a single hashtable. The table is keyed on object pointers; in the event that a pointer is local, the corresponding entry is an IN entry, otherwise it is an OUT entry.

Although hashtables have reasonably good performance, an update operation is clearly still expensive. In the best case, the pointer is hashed; the appropriate entry-key is read from memory; the pointer is compared against the key; the reference count is read from memory; the count

is incremented; the count is written back to memory. This accounts for at least three memory references – two word-reads, one word-write – on an inter-area pointer-copy, which effectively quadruples the memory bandwidth cost of the copy at both source and destination.

We can ameliorate this cost somewhat by providing a cache on each node dedicated to assisting IN and OUT bookkeeping; the per-node cache arguments the per-area hashtable, in the following manner: The cache is keyed on [pointer,area] tuples, and contains IN and OUT entries much like an area's hashtable; the area component of the key specifies which which area a cache entry is associated. An increment or decrement request which hits in the cache requires no memory access. A cache entry whose reference count goes to zero is invalidated. An increment or decrement request which misses in the cache causes an invalid cache entry to be allocated to handle the request – note that this means that counts in the cache may be negative! In the event that no invalid entries are available to handle the request, the cache raises an interrupt; software merges the cache entries into the various hashtables for which they are intended, and invalidates the entire cache.

Merging an OUT entry in the cache with an OUT entry in the hashtable may uncover a strange situation: the two entries could refer to different copy-tree parents. In this event, in addition to the count from the cache being added to the count in the hashtable, a decrement message is sent to the area named in the cache's entry.

As long as a small number of pointers are being bandied about inter-area at any one time, the cache should be very successful at minimizing the cost of IN/OUT bookkeeping.

5.3.5 Design rationale

A number of facts and assumptions underlie my belief that indirect reference counting is a good mechanism for tracking inter-area references.

I believe that inter-area pointers will be relatively rare; many computations will be able to operate within a single area per node, thus avoiding intra-node inter-area references. This assumption leads me to believe that IN and OUT sets will be fairly small, and that a cache of fixed size will be able to manage most IN and OUT set operations without requiring software assistance.

I believe that inter-area cycles will be exceedingly rare. In particular, I expect the most common form of inter-area reference to be the distribution of pointers to a striped object to each of the areas through which the object is striped, in order to perform parallel operations upon the object. When a parallel operation has completed, the various pointers to the object will, in all likelihood, be gone, allowing the reference count to drop back to zero on the object. (The allocation and garbage-collection of striped objects is discussed in detail in Section 5.8.)

Finally, I believe that inter-node communications bandwidth will be in short supply. IRC is network-friendly in that it requires exactly two messages per inter-area pointer copy [1], whereas any incremental GC scheme which requires explicitly traversing inter-area pointers will require N+1 inter-area messages for an inter-area pointer that lasts through N garbage-collections.[2]

By implementing and evaluating the IRC scheme in simulation, I will experimentally determine the accuracy of these beliefs as part of the final thesis.

[1] I should reiterate here that IRC will not discover cyclic garbage structures; another scheme will be needed to discover such structures, and that will add to the overall messaging cost of system-wide garbage collection.

[2] By first compiling a list of remote pointers on the area, a GC could send only one message per remote target, rather than one per copy of the pointer on the area; of course, compiling that list requires work quite similar to the compilation of the OUT list for IRC.

5.4 Collecting cyclic garbage

Unfortunately, IRC does not recover space occupied by inter-area garbage cycles. Thus, to collect inter-area cycles, we perform a rather unusual marking pass known as indirect marking [Piq95] to identify truly live data; the unusual aspect of this scheme is that marks are propagated between areas not from pointer to target, but rather from child to parent in the IRC copy-tree.

Marking is an expensive operation, and one that we hope to have to perform relatively infrequently. In this section I provide a brief, informal description of how this marking process works. In particular, I begin by explaining how to perform a global marking operation covering all areas, and then explain how the system could be extended to allow independent marking of subsets of areas.

This algorithm obeys invariants described in [DS90] that guarantee termination.

5.4.1 Global marking

Indirect marking [Piq95] performs inter-area markings on the copy-tree, rather than the inter-object links themselves. This approach has the important feature that it not only discovers which objects are live, but also which portions of copy-trees are live; dead objects and dead branches are pruned simultaneously.

To begin with, each area starts a local marking-thread or threads to mark from its local root objects. Within an area, mark bits are propagated across pointers to mark all reachable objects. When a local marking-thread comes across a pointer to a remote area, it looks up the pointer's OUT entry; if the entry is not already marked, it marks the entry, and spawns a remote-marking-thread in the parent area listed therein.

The new remote-marking-thread marks the appropriate IN/OUT entry in its area, and then starts marking from the pointer in question. If the area isn't the pointer's home area, the remote-marking-thread does little more than spawning another remote-marking-thread on the next area up the copy-tree; if the area is the pointer's home area, the target object is used as the basis for further marking.

Local marking-threads don't terminate until all of their remote-marking-thread children have terminated; similarly, a remote-marking-thread won't terminate until any children it may have generated have terminated.

Thus, when all the local marking-threads of an area have terminated, every object reachable from the roots of that area has been marked; when all areas' local marking-threads have terminated, all reachable data in the system has been marked, and all copy-tree entries for which some copy of the pointer is still reachable have also been marked.

At that point, any unmarked OUT entry is eliminated and a decrement message is immediately sent to its parent in the copy tree; any unmarked IN entry is eliminated without further ado. (Any decrement messages from an eliminated entry to an eliminated entry may safely be ignored.) This operation eliminates the IN and OUT entries that were pinning garbage cycles into place, and adjusts the outstanding copy counts of surviving data appropriately. At this point, all objects that were in garbage-cycles are free to be reclaimed by area-local GCs.

5.4.2 Subset marking

The problem with a global marking pass is that it requires touching every area. If most inter-area cycles turn out to be between areas on the same node, performing a whole-heap marking to collect them is clearly overkill; we would like a method for marking a subset of areas without having to

reference areas outside the subset. [LQP92] and [Piq96] describe schemes based on dynamically grouping areas by adjusting (copies of) reference counts to effectively create larger areas. This approach allows the dynamic formation of arbitrarily overlapping subsets.

I propose a slightly simpler approach in which we maintain multiple counters simultaneously representing different groupings, all of which are maintained in real-time to avoid the delay of stopping to form dynamic groups. Furthermore, we restrict subsets to either be non-overlapping, or one must be strictly contained within the other; this means that we only need to update the counter of the smallest containing subset when we copy a pointer.

In particular, an IN/OUT entry has a counter for its node, and for the whole system. When a pointer is copied from its home area to an area on a different node, the SYSTEM counter in the pointer's IN entry is incremented; if the pointer is copied from its home area to an area on the same node, then the NODE counter is incremented *instead of* the SYSTEM counter.

Now, to mark a node, we use as roots the IN and OUT entries with non-zero SYSTEM counters. This marking pass discovers any intra-node garbage cycles whose copy-trees are also intra-node, but it does not discover cycles in which either part of the cycle itself or part of the copy-trees for the objects are outside of the node's set of areas.

Note that some links in a copy-tree may pass out of the subset (node) being GCed; mark messages need not be sent over such links.

At the cost of further enlarging the IN and OUT entries, the subset granularities could be further varied, for instance providing independently markable clusters of nodes within the system, and independently markable clusters of areas within a single node. My intuition, however, is that in general, two levels of hierarchy – individual node, and whole system – is the correct tradeoff. [3]

Rather than enlarging each IRC hardware cache entry to contain multiple counters, we can instead key the cache on a tuple which includes the type of the counter being adjusted.

In subsequent sections I will assume that IN and OUT entries have distinguished NODE and SYSTEM counters, in order to allow independent marking and related operations on individual nodes.

5.5 Forwarding pointers for object relocation

Up to this point, I have mandated that an object in one area which is reachable from another area may not be moved by garbage-collection. In this section, I remove that restriction by introducing the forwarding pointer, a hardware-recognized construct that forwards memory references from one location to another. Using forwarding pointers allows us to relocate objects without having to immediately update all outstanding pointers to them; the most important advantage of this is that we need not indirect inter-area pointers through entry vectors to accommodate object relocation, but may instead have direct inter-area pointers that (usually) point directly at their target data.

The application of forwarding pointers is only half the battle however; getting rid of forwarding pointers after they have been created, thus reclaiming the memory they consume, is the other half, and I present some solutions to that problem as well.

5.5.1 Intra-area forwarding pointers

Intra-area object migration is useful for compacting garbage collection, which generally improves data locality. Any object which is only reachable from within the area in question may be migrated without constraint, as long as all pointers to it are updated in the area simultaneously.

[3] Actually, I will add a third, dynamic category when I discuss deactivating areas in Section 5.6.

Using forwarding pointers, however, we can avoid the need to instantly update all other pointers in the area. Instead, when we migrate an object intra-area, we can fill the space it is vacating with forwarding pointers referencing its new location; we call the resulting cluster of forwarding pointers a forwarding object. This may ease the implementation of concurrent area-local garbage collection. When the local GC is complete, the forwarding objects may be discarded.

Perhaps more importantly, however, forwarding pointers allow us to migrate objects reachable from other areas by leaving a forwarding object behind when the object is moved. Note that when migrating an object intra-area, if an IN entry existed for the object pre-migration, that entry is preserved, unchanged, to point to the forwarding object with which the object is replaced. Unfortunately, the forwarding object cannot be discarded as long as remote areas still have pointers to it (as evidenced by the continued existence of an IN entry for the old location.)

5.5.2 Inter-area forwarding pointers

Inter-area object migration has various applications in terms of improving locality, but I will defer serious discussion of the application until Section 6.

Inter-area object migration is much like intra-area object migration: a forwarding object needs to be left behind if there are outstanding pointers to the object that can't be patched at the time of the migration. However, since the forwarding object points to the migrated object in a remote area, the appropriate bookkeeping needs to be performed to keep the IN/OUT sets up-to-date.

We assume that in the case of intra-node forwarding pointers, the memory system finishes traversing forwarding pointer chains for one request before answering the next request, thus preserving the order of memory access requests.

Inter-node forwarding pointers, while straightforward in implementation, have unfortunate semantic effects: two accesses of the same target object can become reordered by virtue of the first of them going through a forwarding pointer on a different node while the second goes directly to the target node. This means that, for instance, code which performed two writes through potentially aliased pointers would have to wait for the first write's completion to be confirmed before proceeding with the second. For this reason among others, in Section 6 I will discuss schemes which never expose inter-node forwarding pointers to user threads.

5.5.3 Eliminating forwarding objects

The trouble with migrating objects reachable from other areas is that the forwarding objects left behind cannot be reclaimed until the pointers in other areas have been updated to point at the real object's new location. The scheme in this section fixes that problem. A new copy-tree is constructed, targeting the object's new location, as pointers are adjusted to point there.

Eliminating forwarding objects happens in two phases: in the first phase, each area discovers which of its OUT entries' targets are now forwarding objects; in the second, the area rewrites each local instance of a remote pointer with its forwarded target.

The first phase can occur in two ways. First, it can be piggy-backed on a cycle-collection operation; if a remote-marking-thread is invoked on a pointer which turns out to target a forwarding object, when it notifies its parent-thread of its termination, it also returns the forwarding object's destination.

Alternatively, a dedicated forwarding-pointer-elimination phase may be undertaken, in which each area processes its OUT list, and sends an UPDATE-REQUEST message to the area containing the OUT pointer's parent entry in the target object's copy-tree. Areas containing OUT entries,

upon receiving UPDATE-REQUEST messages, may opt either to respond immediately with NO-UPDATE, or with whatever information is currently stored in the OUT entry; alternatively, they can propagate the request toward the copy-tree root. Areas containing IN entries, upon receiving UPDATE-REQUEST messages, respond with any forwarding data for the target objects; the response information is propagated back up the copy-trees.

In either case, if messages come to an area specifying new targets for OUT pointers, the updated targets are written into the old OUT entries, each of which have a special FORWARDING flag set; then, a new OUT entry is made for the object's new location. The new OUT entry's copy-tree parent area is the same as the old OUT entry's copy-tree parent area, because regardless of how the notification arrives at this node, it has climbed the new copy-tree from the root – i.e. an appropriate entry is guaranteed to exist in the parent area. The appropriate counter in the parent area's copy-tree entry is incremented when the notification is sent to the child node.

In the second phase, we simply extend area-local GC to examine the OUT entry for every remote pointer discovered; if an OUT entry has a set FORWARDING flag, the GC process replaces the old pointer with the new one.

Since on each elimination phase, a forwarding pointer in a forwarding pointer chain will be updated to point to the next target, this scheme effectively implements pointer-jumping: a chain of N forwarding objects will collapse after log(N) elimination phases.

5.6 Deactivating areas to reduce paging

One trouble with the GC schemes described thus far is that global marking and global forwarding-pointer elimination both require bringing every area in the system into core memory. This is a major problem when the heap is, as we expect it to be, very large due to the presence of huge persistent data-structures: GC should seldom need to examine such data structures when they're not in active use, and thus dragging them into memory is terribly inefficient.

To solve this problem, I introduce the notion of "active" and "inactive" areas. Areas are made "active" or "inactive" depending on their current level of use. Only active areas participate in normal GC operations. System-wide marking uses all pointers out of inactive areas as roots in GCing active areas.

Inactive areas have the additional property that, unlike active areas, they don't pin forwarding objects in place.

As long as a rough correspondence is maintained between being "active" and being in-core, little paging is required to perform precise garbage collection. Deactivating an area incurs the most significant overhead in the entire scheme, as the entire contents of the area may need to be scanned in order to set an INACTIVE bit in outgoing pointers. When an area is activated, INACTIVE pointers are lazily activated as they are traversed.

5.6.1 Deactivating an area

To handle area deactivation, I add a WORLD counter to IN and OUT entries, which creates another, larger-granularity level in the cycle-collection hierarchy discussed in Section 5.4.2. We also add a single-bit INACTIVE flag to OUT entries.

When an area is deactivated, a deactivation thread is started which performs several tasks.

First, the thread scans the area's OUT set. Any entry whose INACTIVE flag is set is left alone. For each remaining entry, the following sequence of events takes place:

1. If the OUT entry has a parent in the IRC tree, the thread sends a DEACTIVATING message to the parent which causes it to be removed from all subsets except the world subset by decrementing the appropriate active-reference counters. (This is essentially the algorithm suggested for removing areas from groups in [Chr84].)

2. The thread sets the INACTIVE flag in the local OUT entry.

Finally, the thread scans all pointers out of the area and sets their INACTIVE bits; this step can be skipped if no OUT entries were active, or sped up by marking pages containing active remote pointers. At this point, the area is entirely deactivated and can be swapped out to disk.

Using this algorithm, every object reachable from an inactive area is denoted by a non-zero WORLD counter in at least one OUT or IN entry. Marking operations use such entries as roots.

5.6.2 Normal pointer manipulation

Pointer manipulation is not substantially changed by the addition of inactive areas. A pointer into an inactive area may be freely copied both within and between active areas. Intra-area copying is obviously entirely free; for inter-area copies, the copy will simply build up the IRC tree as usual, requiring no access to the inactive area.

There are a few of cases in which IN or OUT entries in an inactive area need to be updated when an OUT entry in an active area has the inactive area as its parent in its copy tree: when an area-local GC discovers that an OUT entry in the active area is ready to be destroyed; when the active area is deactivated; or when an inactive OUT entry in the active area is reactivated.

However, we do not have to swap the inactive area in each time! Instead, update are written into a node-local log; when an inactive area is reactivated, the log is scanned for updates to that particular area. Log-entries for a particular area may be stitched together in the log with pointers to increase searching efficiency, at the cost of increasing the size of each log entry. The head of the log may reside in-core. I will make further use of the log in the section on heap persistence, Section 7.1.

5.6.3 Garbage collection without touching inactive areas

Performing a node- or system-wide marking pass without touching inactive areas is straightforward: a copy-tree entry with nonzero WORLD counter is treated as a root for all purposes.

Note that a garbage cycle which goes partially through inactive areas will never be collected, since portions of the cycle are simply used as roots during the system-wide mark. The solution to this problem is to run an incremental marking algorithm such as the one described in [LL92] as a very low-priority background process using an independent set of MARK bits. My heap implementation for this thesis will not include this slow GC, but the thesis text will include a more detailed description of its mechanism.

5.6.4 Reactivating an area

Area reactivation is a lazy process. The area is marked reactivated when it is first accessed (i.e. data is read or written; updates to meta-data – i.e. decrementing counts in IN/OUT entries – don't force reactivation.)

When a pointer with a set INACTIVE bit is read from memory, a trap handler is invoked which looks up the pointer in the OUT set.

If the OUT entry does not have a set INACTIVE flag, the pointer is immediately overwritten with the updated information from the OUT entry, and normal execution resumes.

If, on the other hand, the OUT entry for the pointer still has a set INACTIVE flag, the handler sends an ACTIVATE message to the entry's parent in the IRC copy-tree, which causes the WORLD counter to be decremented. The response includes forwarding information if the object has migrated since the area was deactivated. The sending area updates its OUT pointer with any forwarding information, setting the FORWARDING flag if appropriate, and clearing the INACTIVE flag. Next, the handler writes the activated pointer into the slot in memory from which the inactive pointer was read. Finally, normal execution resumes.

Because pointer activation is lazy, only as many pointers as needed are activated; this makes deactivating an area after accessing only a couple of objects fairly cheap. Area-local GC need not activate inactive pointers; however, if an area is active long enough to participate in an inter-area marking operation, all its pointers will be activated by the marking process.

5.6.5 Forwarding lists

A potential problem with inactive areas is that their OUT pointers will never be updated if their target objects are migrated; one could view this as pinning forwarding objects in place indefinitely. However, since our protocol requires a reactivating OUT entry to examine a IN entry to discover if any migration has happened, we can actually throw away forwarding object which is reachable only by inactive areas, as long as we do two things: we must preserve the IN entry identifying the forwarding object until its WORLD counter goes to zero; and we must not reuse address space for which forwarding IN entries exist. IN entries containing forwarding information but not backed up by actual forwarding objects form a sort of forwarding list that replaces the set of forwarding objects "pinned" in place by pointers from inactive areas.

Note that the second constraint – not reusing address space containing forwarding entries – should be simple to obey in a machine with sufficiently large address space simply by never reusing any address space; in any event, the background incremental global garbage collector which eliminates garbage cycles passing through inactive areas can also incrementally squeeze out forwarding pointers, eventually cleaning regions of address space polluted with forwarding list entries.

5.7 Deciding what to do when

Up to this point, I have presented a variety of mechanisms for collecting garbage on the system in a paging-friendly fashion; while I have explained in general what each mechanism is designed to achieve, I haven't addressed exactly when each mechanism should be invoked. In this section, I present some high-level heuristics for applying each mechanism; working out the details to make these heuristics successful will be an important part of the final thesis.

5.7.1 Area-local GC

Area-local GC should be triggered in active areas under a handful of circumstances.

One trigger is simply a timer that triggers after a certain time since the last area-local GC. This is simply to collect any objects that may have become garbage since the last area-local GC, in order that more physical memory might be freed up for use in other areas which might be experiencing more demand than the current area. The exact period of this timer is a matter for further thought and experimentation.

Another trigger is running out of memory allocated to the area; although an area's address space is large, little of it is backed by physical memory or swap at any given time. Before allocating more backing store, an area-local GC should be attempted.

A third trigger is area deactivation; an area-local GC should be performed just before an area is deactivated, both to minimize the amount of data that is actually paged out in the area, and to eliminate any garbage pointers that might pin garbage objects in other areas. Additionally, this area-local GC can be used to discover the locations of all out-of-area pointers for deactivation.

Finally, of course, area-local GC is used as a component of some of the more comprehensive GC operations.

5.7.2 Node-local marking

A node-local marking phase should be triggered when an area-local GC is unable to free enough memory R, and finds that more than a certain fraction F of used memory is in use only because of objects whose IN entries' WORLD and SYSTEM counters are zero (i.e. pointers to these objects have not been copied directly from their home area to other nodes, nor been paged out in deactivated areas.) Rather than being simple constants, these parameters may end up being related by an equation.

Node-local marking should also be triggered after a certain time since the last node-local marking phase. This prevents gradual buildup of intra-node garbage cycles.

5.7.3 System-wide marking

System-wide marking should be invoked only very rarely, as it is quite communications-intensive. It is, however, quite preferable to heavy paging activity. Thus, the trigger for system-wide marking should be related to paging. In particular, when one or more nodes determine that they are paging excessively (the definition of "paging excessively" being a parameter open to exploration) they must call for a system-wide marking operation in an attempt to free up any inter-node garbage cycles that might be contributing unnecessarily to the active memory footprint.

5.7.4 Deactivating areas

The ideal time to deactivate an area is after a system-wide marking pass which guarantees that the area is not contributing to any garbage cycles currently in active memory. A very simple heuristic for deciding when to deactivate an area is to mark each area as "unaccessed" after a system-wide marking pass; this mark is cleared if the area is subsequently read from or written to. In the event than an area is still marked as "unaccessed" when the next system-wide marking pass is invoked, the area is deactivated immediately following the conclusion of the marking pass.

Areas for which more than a certain portion of their data is directly reachable from area-roots could be deactivated after a node-local marking phase concludes with relatively little risk of accidentally pinning global garbage cycles.

5.8 Managing striped objects

In this section, I finally discuss how to extend the management schemes already described to handle the allocation and garbage-collection of striped objects.

5.8.1 The full definition of an area

Up until now, I have been speaking of an area as being composed of a single contiguous piece of address space, all of which resides on one processor. Now I expand the definition considerably: an area consists of a set of address spaces, one in each of the possible striping-granularity spaces. An "intuitive" explanation turns out to be rather confusing, so I offer a formal definition first, followed by the intuitive explanation

First, recall that an address is of the form $[S, V_h, N, V_l]$ where S is the striping-granularity count which selects the position of N within the address; V_h is the high bits of the virtual address; N is the identifier of the node responsible for this particular word of data; and V_l is the S low bits of the virtual address.

Formally, an area is defined by a range $[L, H]$. The area contains all addresses $[S, V_h, N, V_l]$ such that $L \leq V_h V_l < H$ where $V_h V_l$ is the concatenation of V_h and V_l.

Informally, this means that an area is responsible for an equal number of words in each striping-granularity space, but that the words that an area is responsible for in a given space are not necessarily contiguous – they consist of only those words within a larger address range which happen to fall on the area's home node.

I refer to two areas on different nodes with the same bounds $[L, H]$ as correlated areas; I refer to two areas on the same node as co-resident areas. I refer to a set of correlated areas with one from each node as a complete correlated set (CCS) of areas.

5.8.2 Allocating striped objects

A striped object is allocated entirely within one CCS of areas. Within each CCS, one area is chosen to manage allocation for each striping-granularity; by putting one area in charge, objects at that granularity will be densely packed, whereas giving each of the areas charge of allocations within some subset of each stripe space would produce sparsely packed objects, resulting in poor utilization of physical memory.

In order to manage striped objects, we add a new set to each area, the STRIPED-OBJECTS set. The STRIPED-OBJECTS set will have an entry for every striped object whose first word lies in the area, in which case I call the area the object's head area.

When a thread running in some area wishes to allocate a striped object, it sends a request to the area in its CCS responsible for that striping granularity.

Upon receiving the request, the allocating area simply counter marking the end of the allocated region of the address space; it then sends an "initialize" message to each area through which the newly allocated space passes; finally, when initialization is complete, it returns the address of the space thus allocated to the requesting thread.

An area which receives an initialize message for a particular region may need to do several things. First, it always advances a local end-of-striped-region pointer to encompass the newly allocated object. Next, if it happens to be the object's head area, it makes an entry in the area's STRIPED-OBJECTS list; unless the head area is also the area requesting the data, it creates an IN entry for the object with the appropriate subset reference count set to 1. Finally, if the area happens to contain any padding words at the tail of the striped object, it writes the object length into those words.

When a pointer to the requested region is returned to the area making the initial request, unless the area is also the new object's head area, an OUT entry is created for the pointer which corresponds to the IN entry created in the head area.

At this point allocation is complete, and pointers to the newly allocated object may be used normally.

Combining Allocation Requests: Striped object allocation requests may be made with combiners (see Section 4.1.5). In this case, if multiple requests for space in the same stripe space are issued around the same time, they can be merged. The returned space can then be split up to satisfy each of the requests. However, in this case, since the allocating node no longer knows where all the beginning and endings of objects will actually be, the requesting nodes must take responsibility for sending the appropriate initialize messages. This is the scheme I actually propose to use.

Allocating Aligned Objects: Allocating an object which is aligned with another object – i.e. the Nth element of the new object is on the same node as the Nth element of the old object – is performed by padding the allocation as much as necessary to put the new object's head at the same node as the old object's head. This strategy may well result in huge amounts of initially wasted memory; we leave it to the garbage-collection phase to improve memory utilization.

5.8.3 Bookkeeping with striped objects

For all GC bookkeeping purposes, a striped object is treated as if it resided in its head area.

5.8.4 Garbage collecting with striped objects

Garbage collecting with striped objects is fairly straightforward. I will discuss three distinct GC operations: area-local GC, node-level marking, and system-wide marking.

Area-local GC simply uses any stripes passing through as roots for the GC. Since an area tracks the valid region of each striping space, it can simply use every word in that region as part of the root set, without worrying about which parts of it are stripes of what object.

Similarly, since the heads of striped objects may be on different nodes, node-local marking uses stripes as roots for marking.

If we were to allow arbitrary areas to be deactivated, we would have to impose this limitation even on system-wide marking – since the head areas of striped objects might be deactivated, we would have to continue to use stripes through all active areas as roots for marking. Thus, we might never eliminate intra-node cycles involving striped objects, even when 95% of the correlated areas through which the striped objects ran were active.

To avoid this problem, when system-wide marking begins, for every currently active area, the corresponding complete correlated set of areas is activated. This guarantees that the head areas of all striped objects passing through any active areas are, themselves, active.

This guarantee allows system-wide marking to omit object stripes from the root set. Instead, when a striped object is discovered to be reachable from a real root (or an entry from an inactive area), its head area spawns a remote-marking-thread in all of the correlated areas through which it passes; each of these threads uses the stripes of the object passing through its area as a basis for further marking.

5.8.5 Garbage collecting striped objects

A striped object may be garbage collected if it meets these conditions:

1. It has no IN entry – i.e. there are no remote-node pointers to it outstanding.

2. An area-local GC discovers no local pointers to it (except possibly self-referential pointers.)

These conditions are fairly obvious, and are, in fact, the same conditions as for garbage collecting an unstriped object. However, the mechanism by which a striped object is actually garbage-collected is somewhat different.

Discovering garbage striped objects

Discovering garbage striped objects actually happens as a result of area-local GC. To actually discover garbage striped objects, an area first clears a REACHABLE bit on each entry in its STRIPED-OBJECTS set for which the corresponding object has no IN entry. The area then performs a local GC which does not include the STRIPED-OBJECTS set as part of the root set; as pointers to local striped objects are discovered, the REACHABLE bits are set in their STRIPED-OBJECT entries. At the end of the local GC, any striped object whose STRIPED-OBJECTS entry does not have a set REACHABLE bit is garbage.

Compacting out garbage stripes

Upon discovering that some striped objects are unreachable, an area sends a message to the area which allocated those objects. Allocator areas maintain a garbage-space list into which they merge these reports of garbage. When the garbage density gets too high, it is time for a compaction.

Compaction is a tricky business, however. For each address of the form $A = [S, V_h, N, V_l]$, we obviously have to leave S, the striping granularity, alone.

We also need to leave N, the node identifier, alone, for the following reason: we want to compact data in such a way that only intra-area data copying is necessary; since striped objects are likely to be huge, we really don't want to have to move their data sets around intra-node. Additionally, if we start moving these objects around inter-node, we have to explicitly remember object alignment specifications and maintain them when we perform such migrations. Thus, we leave N fixed.

Adjusting V_l consistently for an object which is striped over more than one node is impossible because adding an offset will cause N to roll over for at least some of the addresses spanned by the object. Thus, compaction may only adjust V_h. (Note that while in theory adding an offset to V_h could cause S to roll over, in practice, the address space denoted by V_h will be large enough to prevent this from actually becoming an issue.)

At the beginning of the compaction operation, the allocator area has collected a list of garbage regions, whose inverse defines a list of live regions; note that the allocator area can not generate a list of individual live objects, but rather only live regions which potentially consist of multiple live objects.

In order to perform a compaction, we first construct a table mapping the current set of live regions into a more compact form in a new address space; this table will be broadcast to every area in the complete correlated set of areas so that each area can actually copy live stripe-data from the old space into the new, more compact space. Note that the copy operations always leave forwarding pointers behind, which will eventually be eliminated using the normal forwarding-pointer elimination mechanisms.

To construct the mapping table which guides the actual compaction, we use a simple greedy algorithm. First, we sort the regions by the $[N, V_l]$ component of the address of their first words; we place the sorted list in a binary tree so that we can perform binary search in $\log(N)$ time.

Now, we take the first live region and allocate it a new address range at the top of the new address space; we remove it from the binary tree. Then we search the tree for the next live region

whose head most closely abuts the first region without overlapping it, remove this region from the list, and allocate it its spot in the new address space. This process is iterated until all live regions have been allocated new space. The sort operation takes $O(N \log(N))$ operations for N live regions, as does the subsequent region-by-region selection; if this cost turns out to be prohibitive, I may attempt something simpler.

Chapter 6

Object Migration

In the previous chapter I introduced forwarding pointers as a mechanism supporting compacting garbage collection by allowing objects to be moved without immediately having to update all pointers to those objects. In this chapter I discuss mechanisms and heuristics for further improving locality by migrating objects inter-area and even inter-node.

6.1 Migration timing and basic mechanics

Moving an object between areas on the same node is a simple process: the object is copied to its location in the new area, and its old location is filled with forwarding pointers to its new location. An IN entry is created for the object in its new area, and an OUT entry is created in the area from which the object was copied.

Moving an object between nodes is more complicated – not because the same mechanism wouldn't work at first glance, but because, as mentioned in the previous chapter, the presence of inter-node forwarding pointers can cause writes from a single thread of control to be reordered by virtue of some of them bouncing through multiple nodes to reach the final destination; I do not wish to impose this burden upon the programmer or compiler.

Thus, when an object is selected for migration between two nodes, an entry is made in a MIGRATION set in its home area; the entry lists both the object and its eventual target area. The target area adds the size of the target to a local INCOMING counter in order to keep track of how much data is going to suddenly get added to the area.

Objects in the MIGRATION set are migrated during an inter-node forwarding-pointer-elimination phase. When one of these operations is about to be performed, first all scheduled inter-node migrations are actually performed, leaving forwarding pointers behind; then, when the forwarding-pointer-elimination operations occurs, the newly-created forwarding pointers are squeezed out again before any user code is exposed to them.

6.2 Explicit (manual) migration

From time to time a programmer may know exactly to which area she wishes a particular object to be relocated; in this case, she may invoke a simple system routine which performs the migration if it is an intra-node request or schedules it for migration if it is an intra-node request.

6.3 Heuristics for automatic migration

6.3.1 Ineligible objects

Although moving objects closer to where they are referenced improves locality, moving objects inter-node can have a potentially bad side-effect in a parallel system by pulling together on a single node data which was deliberately distributed over many nodes in order to allow parallel operations upon that data.

I expect parallelism to be expressed most frequently via operations over all elements of striped objects, and thus we will not migrate objects reachable from stripe elements. On the other hand, unless the programmer specifically anchors non-striped objects, we will tend to migrate them in order to improve locality of reference.

To be specific, the following classes of object are immune to automatic migration:

- striped objects
- objects reachable via an in-area path from an area's true root set
- objects reachable via an in-area path from in-area stripes of striped objects
- objects marked by the programmer at allocation-time as ANCHORED

Striped objects aren't subject to migration because moving them inter-node is prohibitively expensive, and moving them intra-node is fairly pointless given their distributed nature.

Objects reachable from an area's true roots (i.e. live threads' registers, etc.) aren't subject to automatic migration because such objects are likely in use in the local area, and thus migration is likely to worsen locality of reference.

Objects reachable from stripes in the area aren't subject to automatic migration because they are likely to be accessed when parallel operations over the striped objects start threads in each of the correlated areas, and thus there will likely exist an in-area thread at some point which will access the stripe-reachable objects.

Finally, a programmer might choose to mark some objects as anchored in specific areas in order to keep them distributed, rather than having them coalesce to a single area.

All other objects are fair game for migration and have their IN entries marked as such by area-local GC. The hard part is figuring out to which area a candidate object should migrate.

My migration heuristics will focus on migrating candidate objects only into areas in which they will be reachable from one of the unmigratable object classes listed above; this will insure that, for instance, an inter-area garbage cycle doesn't migrate in circles wasting communications and memory bandwidth.

6.3.2 Migration offers

Objects which are candidates for migration will, by definition, appear in their host area's IN set; however, there is no indication in an IN entry as to which remote areas might be able to reach these objects, or which of those areas would be appropriate migration targets. Thus, the possibility of object migration begins with a migration offer from a remote area to the area containing a potential candidate object.

Deciding whether or not to make a migration offer is a simple process: an area with a capability for a remote object can tell from the capability how large the remote object is; if the remote object would fit in the area, the area makes the offer. This size-comparison should not be performed on

every reference, but rather on a very small, randomly selected, subset. This approach not only keeps the amortized per-reference overhead down, but also increases the likelihood that the limited amount of extra space in the area will be filled with frequently-accessed objects, rather than simply the first few migratable objects referenced.

A migration offer is piggy-backed on an inter-area reference to a remote object. This approach insures that objects which accept a migration offer are migrated to an area which is actively referencing them, and that objects which aren't being actively accessed at all are not migrated – in short, memory and communications-network bandwidth are never wasted on pointless migrations. The piggy-backing cost is small, taking the form of a single bit which is set when the reference is also a migration offer.

6.3.3 Migration scheduling

Upon receiving a migration offer, an area checks to see if the object is a candidate for migration; if it is a candidate, and is not already scheduled for migration, then the area schedules the migration and sends an acceptance message back to the offering area to confirm the migration. Otherwise, the area sends a message back to the offering area denying the migration request. These return messages take the form of a single bit piggy-backed on the confirmation of the access.

Note that the scheme as described has a serious limitation: only objects which are directly reachable from remote areas are ever scheduled for migration. Under this restriction, migrating a length N linked list would take N migration passes. We solve this problem by making migration-scheduling contagious, as follows:

When an area-local GC is performed, any objects which are found to be reachable only from an object already scheduled for migration are tentatively scheduled for simultaneous migration; entries for each of these objects are added to a list depending from the originally-scheduled object's migration entry, and the total size of the collection of objects is recorded in the first entry.

This adds a little complexity to the migration protocol. At migration-time, the target area must be queried not only for an address to which to write the migrated object, but also for space for the additional objects reachable from it. In the event that it has insufficient space, the target area may opt to deny the request for additional space, or to provide a smaller quantity of space, thus only enabling the migration of a subset of the objects.

Chapter 7

Persistence

7.1 Consistency and crash-recovery

It is important to carefully manage updates to any persistent heap in order to provide a consistent view of the heap in the event of a system failure. I solve the problem by occasionally checkpointing the heap, which includes the state of all threads.

To begin with, if all areas are inactive and stored on disk, then we have a consistent view of the entire heap.

When an area is made active at time T, a new region of disk is allocated to back the transient version for paging/swapping, leaving the old view of the region intact.

Modifications to inactive regions consist of updates to IN and OUT entries; rather than provide transient pages for all inactive areas, we simply rely on the node-local log described in Section 5.6.2, where each IN/OUT-entry modification is entered, along with a time-stamp, in the log.

A checkpoint operation consists of the following sequence of events, which uses a two-phase commit mechanism:

1 A designated or randomly-selected coordinator node broadcasts a CHECKPOINT message to all nodes.

2 Any messages currently in the communications network are delivered, and all user threads are suspended at all nodes. Area-local garbage collection is halted.

3 For each node:
   - A snapshot is made of all dirty pages of in-core areas by writing them to their transient backing pages.
   - A PREPARE entry is made in the node-local log describing the location of the transient backing pages; any in-core portions of the log are flushed to disk.
   - The coordinator node is notified that the node has finished the prepare phase.
   - User threads are resumed, but are not allowed to cause paging that would overwrite the on-disk transient backing pages.

4 The coordinator node writes a COMMIT entry to its log when all nodes have signaled their completion of the prepare phase. At this point, the checkpoint is committed for crash-recovery purposes.

5 The coordinator broadcasts a COMPLETION message to all nodes.

6 Each node now simply copies the snapshots of dirty pages (i.e. the transient backing pages) to the persistent locations of those same pages; when a node is finished, it writes an UPDATED message into its local log.

7 User threads may now cause paging normally.
8 Area-local garbage collection is resumed.

We can use *lazy snapshotting* to reduce the period during which user threads must be suspended. When we make the snapshot, what we really do is mark all pages in all areas as read-only. We then allow computing threads to resume while we start writing pages to disk; if, a computing thread attempts to write to a read-only page, a trap handler is invoked which insures that that page has been written out before marking the page writable and allowing the computation to resume. This approach allows computation to proceed while snapshotting is going on.

Crash Recovery: In the event of a system crash, each node scans its local log for COMMIT entries; the most recent COMMIT entry found defines the last successful checkpoint. If an UPDATED entry follows the COMMIT entry, the node may simply restore each area from its persistent storage area; otherwise, the node copies the data from the transient backing pages described in the PREPARE entry into their spots in the persistent storage pages, writes an UPDATED entry, and then restores each area from the persistent space.

Log entries specifying updates to IN and OUT entries which are time-stamped prior to the most recent committed checkpoint are retained; those time-stamped after the most recent checkpoint are discarded.

7.2 Distinctions between persistence and transactionality

It is important to note that there is a distinction between persistence and transactionality, and that where transactionality is the desired property, this persistent heap is not sufficient. This heap is persistent in the sense that in the face of a crash, it can be restored to the state as of the previous checkpointing operation, which is likely to be dramatically superior than requiring a complete restart; however, the last few seconds, or perhaps minutes, of computation will have to be re-performed, and any data-structure updates during that time will have been lost. Thus, an application such as banking in which losing an update is a serious problem would need to use an independent transactional mechanism to insure that no updates were ever lost in the event of a crash.

Note that transactions could be implemented simply by checkpointing the heap after each update; however, checkpointing the heap will generally be far too expensive an operation to perform with high frequency while still retaining good system performance.

Chapter 8

Memory Management Evaluation

In this chapter I briefly describe a few of the metrics by which I shall evaluate the success of the memory-management schemes I have described. In the first section, I discuss the two fundamental requirements of the system; in the second, I discuss specific numerical metrics of interest.

8.1 Fundamental requirements

There are two fundamental requirements for the heap memory-management system: correctness and efficiency.

Correctness: The first and most stringent requirement is that memory-management must work correctly. In particular, memory must be allocated when needed; memory must be recovered when it becomes garbage; and no piece of memory can be recovered or reallocated until such time as it really is garbage. These conditions allow user programs to make correct, safe forward progress in a system with limited physical memory and storage.

Efficiency: The next requirement is that memory-management must work efficiently. This decomposes into several conditions. First, the bookkeeping overhead imposed on user threads must be minimal. Second, allocation must be fast. Third, garbage collection must be efficient both in terms of compute cycles, and in terms of other resources such as bandwidth to secondary storage; in other words, garbage collection must neither monopolize the processor nor cause huge amounts of paging. These conditions are somewhat more subjective than the correctness condition, and will be largely measured in terms of the ratio of the costs incurred by garbage collection verses those incurred by user computations.

8.2 Numerical metrics

In this section I present a handful of numerical metrics which I intend to collect and compute as bases for evaluating the success of my memory-management scheme at meeting the fundamental requirement of efficiency. Each number will be computed independently for each benchmark application run with a variety of different garbage-collection parameters (including no garbage-collection at all), and for groups of benchmark applications running simultaneously.

The metrics listed here are nowhere near a complete set; they are merely the tip of the iceberg.

8.2.1 Specific event counts

Each of the following event counts should be made for every thread of control, and ultimately attributed either to user work, garbage collection, or kernel operations.

- Intra-area, inter-area and inter-node reads/writes
- Intra-area, inter-area and inter-node procedure invocations
- Other inter-area and inter-node messages
- IRC hardware cache hits and misses
- Idle processor cycles
- Number of pages transferred to/from secondary-storage
- Number of forwarding-pointers traversed

8.2.2 Average, amortized, and peak values

The following list is comprised of a variety of metrics, many of which are a little more complicated than simply counting events.

- Per-thread memory metrics:
  - Total memory allocated
  - Mean amount of live memory
  - Average amount of live memory
  - Peak amount of live memory
  - Differentiate by striped and non-striped objects
- Average work (number of inter-area and inter-node messages, number of instructions, etc.) per allocation/deallocation
- Average cost of references (including bookkeeping and paging costs)
- Average and mean latency between when data becomes garbage and when it is collected
- Average and mean amount of heap that is garbage at any given time

Chapter 9

Schedule of Work

The work I propose to do breaks down into roughly five sections: implementing the synthetic workloads; implementing the "hardware" simulator; implementing the memory-management schemes; measuring and evaluating the performance of the memory-management schemes; and an additional collection of features which will be implemented and measured only if time permits.

In the following sections, I discuss each of these areas of work; finally, I provide a list of incremental self-imposed deadlines.

9.1 Workloads

Implementing the synthetic workloads breaks down into several components:

- Benchmark selection
- Benchmark implementation in parallel Scheme
- Pscheme formalization/crystallization
- Stack machine code formalization/crystallization
- Pscheme compiler targeting stack machine

I expect the implementation of the various benchmarks to force me to reconsider aspects of parallel scheme; similarly, I expect the implementation of pscheme to force me to reconsider aspects of the stack machine "hardware". Initially I will implement two pscheme benchmarks, which should drive pscheme's description to a useful crystallization. After that, I will write the pscheme compiler. I intend to perform no significant optimizations; indeed, the compiler will be little more than a simple translator.

Combiners are not likely to be implemented in an efficient fashion for this project.

9.2 Hardware simulator

The hardware simulation will be brought up in several stages:

- Basic single-node stack machine simulator
- Basic system software

- Multi-node stack machine simulator
- Memory-utilization visualization tools

The single-node stack machine simulator will initially have unlimited physical memory; it will be a suitable target for non-parallel scheme compiled with the pscheme compiler, but little more. The basic system software will consist of I/O, memory-allocation routines, and thread-switching routines, and will be a prerequisite for any user code to actually run on the machine.

Once the basic system is simulating, I will extend the simulator to handle multiple nodes by adding a very simple network simulator, "hardware" routines to handle non-local references, and system software to handle striped object allocation.

At this stage, I should be able to run multi-node benchmarks on the stack machine simulator – note that these benchmarks will run with no memory-reclamation whatsoever!

An initial version of a per-node memory-utilization visualization tool should be fairly simple; the goal of this tool will be to determine visually how much memory in a given area is live, garbage, reachable from another intra-node area, reachable inter-node, etc. In order to generate these images, the tool will need to perform marking passes over large regions of memory, so it will not be cheap to generate an image; nevertheless, it will ideally be capable of ultimately generating animations demonstrating memory utilization over time. I expect that the tool will mutate as the project progresses.

9.3 Memory management

Memory management breaks cleanly into two sections: garbage collection, and area deactivation.

9.3.1 Garbage collection

Implementing garbage collection will happen in several stages. In addition to implementing the actual algorithms, adding "hardware-support" will require further additions to the simulator code; software-support will require additions to existing allocation code; and visualizing the results will require modifications to the visualization tools. I will implement them in the obvious order:

- Area-local GC (stop-and-copy)
- Indirect reference counting GC (across all areas)
- Subset-based indirect mark-sweep passes

9.3.2 Area deactivation

Deactivating areas requires the implementation of a virtual-memory subsystem for the stack machine nodes; thus, I will implement a VM system providing a fixed, fairly small quantity of physical memory, and a much larger quantity of secondary storage.

Once the VM system is in place, I will implement the relatively straightforward actual deactivation and reactivation mechanisms.

9.4 Measurement and writeup phase

I will draft technical chapters describing each module's implementation concurrently with the implementation process; it is my hope that the chapters in this proposal will not prove too inaccurate to modify quickly to suit the final thesis.

I will make preliminary measurements as I implement each new module, if for no other reason than to verify the correct operation of the performance-monitoring hooks.

Final polishing of the writeup will happen once evaluation is completed, and may overlap with defense.

9.5 "Bonus" work

In the event that the work already described goes faster than anticipated, there are additional aspects of the total system which I will implement and evaluate.

Inter-area object migration: One of the measurements I intend to make is the frequency of non-local references to potentially-migratable objects. If time permits, I would like to go ahead and add object migration, and thus discover to what degree this reduces the frequency of non-local – particularly inter-node – references.

Persistence: Implementing persistence is in itself likely to be a fairly simple addition to the simulation system; however, it is likely to have significant performance implications demanding serious analysis, and perhaps requiring a concurrent (and therefore complex) implementation in order to be tolerable.

Efficient combiners with hardware support: My initial implementation of "combiners" will simply be to execute them sequentially on the target object; if time permits, I will implement a more efficient scheme in which they are dynamically merged in parallel-prefix style in order to reduce hotspotting.

Concurrency: It is my initial intention to write area-local GC and inter-area marking passes in a non-concurrent fashion; any user threads running in or trying to use data in an area being GCed will be suspended for the duration of the GC. In the event that this stop-the-world approach proves to have terrible performance implications, I would like to make some or all of the GC routines partially or fully concurrent.

9.6 Schedule summary

- **December 6:** Two benchmarks implemented in parallel scheme
- **December 10:** Parallel scheme to stack-machine compiler
- **December 17:** Single-node stack-machine simulator
- **December 22:** Multi-node stack-machine simulator
- **January 8:** Vacation ends

- **January 12:** Memory-utilization visualization application; schedule re-evaluation
- **January 19:** Area-local GC (stop-and-copy) implemented
- **February 4:** Global IRC GC
- **February 18:** Inter-area marking-passes
- **March 3:** Virtual memory added to stack-machine simulator
- **March 10:** Area activation/deactivation implemented
- **April 3:** Measurement and preliminary analyses completed
- **April 17:** Thesis defense
- **April 28:** Final thesis submission

Bibliography

[Bev87] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE '87 Parallel Architectures and Languages Europe*, pages 176–87, June 1987.

[BGH+82] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The scheme-81 architecture – system and chip. In *Conference on Advanced Research in VLSI*, pages 69–77, January 1982.

[Bis77] Peter B. Bishop. *Computer Systems With A Very Large Address Space And Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, May 1977.

[Ble93] G. E. Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[Chr84] Thomas W. Christopher. Reference count garbage collection. *Software – practice and experience*, 14(6):503–7, June 1984.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 319–27, October 1994.

[CLT92] Jeff Chase, Hank Levy, and Ashutosh Tiwary. Using virtual addresses as object references. In *2nd International Workshop on Object Orientation in Operating Systems*, pages 245–48. IEEE Computer Society, September 1992.

[CVvdG90] H. Corporaal, T. Veldman, and A. J. van de Goor. An efficient, reference weight-based garbage collection method for distributed systems. In *PARBASE-90 International Conference on Databases, Parallel Architectures and Their Applications*, pages 463–5, March 1990.

[DS90] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1990.

[Fab74] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–12, July 1974.

[FN96] Cormac Flanagan and Rishiyur S. Nikhil. phluid: The design of a parallel functional language implementation on workstations. In *Proceedings of the first ACM international conference on functional programming*, pages 169–79, May 1996.

[GJS96] James Gosling, Bill Joy, and Jr. Steele, Guy L. *The Java Language Specification*. Addison-Wesley, 1996.

[Gol89] Benjamin Goldberg. A reduced-communication storage reclamation scheme for distributed memory multiprocessors. In *Proceedings of the fourth conference on hypercubes, concurrent computers, and applications*, pages 353–9, March 1989.

[GR74] Leonard Gilman and Allen J. Rose. *APL: an interactive approach. 3rd ed.* John Wiley, 1974.

[Hug85] John Hughes. A distributed garbage collection algorithm. In *Functional Programming and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*, pages 256–72, September 1985.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Dynamic Memory Management*. John Wiley & Sons, 1996.

[Joe96] C. F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, January 1996.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The high performance Fortran handbook*. MIT Press, 1994.

[LDS92] B. Liskov, M. Day, and L. Shrira. Distributed object management in thor. In M. T. Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, 1992.

[Lev84] Henry M. Levy. *Capability-based computer systems*. Digital Press, 1984.

[LFPS98] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Programming language design and implementation*, pages 152–61, May 1998.

[*LI86] *The Essential *LISP Manual: Release 1, Revision 7*. Thinking Machines Corporation, July 1986.

[LL92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *12th International Conference on Distributed Computing Systems*, pages 708–15. IEEE Computer Society, June 1992.

[LM86] Claus-Wener Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Proceedings of the 1986 ACM conference on lisp and functional programming*, pages 343–350, August 1986.

[LQP92] Bernard Lang, Christian Queinnec, and Jose Piquer. Garbage collecting the world. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–50, January 1992.

[Mah93] Umesh Maheshwari. *Distributed garbage collection in a client-server, transactional, persistent object system*. Master of engineering thesis, Massachusetts Institute of Technology, February 1993.

[ML95] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Principles of distributed computing*, August 1995.

[ML96] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. Technical Report MIT/LCS/TR 699, Massachusetts Institute of Technology, October 1996.

[Mor98] Luc Moreau. A distributed garbage collector with diffusion tree reorganization and mobile objects. In *Proceedings of the 3rd ACM international conference on functional programming*, pages 204–15, September 1998.

[Mos89] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The mneme project's approach. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 358–74, June 1989.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering (TSE)*, 18(8):657–73, August 1992.

[NAH+95] R. S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J. Maessen, and Y. Zhou. ph language reference manual, version 1.0 – preliminary. MIT LCS CSG 369, Massachusetts Institute of Technology, January 1995.

[Ng96] Tony C. Ng. *Efficient garbage collection for large object-oriented databases.* Masters thesis, Massachusetts Institute of Technology, May 1996.

[Piq91] Jose M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 Parallel Architectures and Languages Europe*, pages 150–165, June 1991.

[Piq95] Jose M. Piquer. Indirect mark and sweep: A distributed gc. In *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 267–82, September 1995.

[Piq96] Jose M. Piquer. Indirect distributed garbage collection: Handling object migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–47, September 1996.

[PS95] David Plainfosse and Marc Shapiro. A survey of distributed garbage collection techniques. In *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 211–249, September 1995.

[PV98] Jose M. Piquer and Ivana Visconti. Indirect reference listing: A robust, distributed gc. In *Euro-Par '98 Parallel Processing*, pages 610–19, September 1998.

[Ran83] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17(1):43–6, July 1983.

[Sab88] Gary W. Sabot. *The paralation model: architecture-independent parallel programming.* MIT Press, 1988.

[San84] Nandakumar Sankaran. a bibliography on garbage collection and related topics. *ACM SIGPLAN Notices*, 29(9):149–158, September 1984.

[SDP92] March Shapiro, Peter Dickman, and David Plainfosse. Robust, distributed references and acyclic garbage collection. In *Principles of distributed computing*, August 1992.

[WK92] Paul R. Wilson and Sheeta V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *International Conference on Object Orientation in Operating Systems*, pages 364–77. IEEE Computer Society, IEEE Press, Sept 1992.

[WW87] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE. Parallel Architectures and Languages Europe*, volume 986 of *Lecture Notes in Computer Science*, pages 432–43, June 1987.