Artificial Intelligence and Language Processing

Jacques Cohen Editor

Improving Locality of Reference in a Garbage-Collecting Memory Management System

ROBERT COURTS

ABSTRACT: Modern Lisp systems make heavy use of a garbage-collecting style of memory management. Generally, the locality of reference in garbage-collected systems has been very poor. In virtual memory systems, this poor locality of reference generally causes a large amount of wasted time waiting on page faults or uses excessively large amounts of main memory. An adaptive memory management algorithm, described in this article, allows substantial improvement in locality of reference. Performance measurements indicate that page-wait time typically is reduced by a factor of four with constant memory size and disk technology. Alternately, the size of memory typically can be reduced by a factor of two with constant performance.

1. INTRODUCTION

Since the late 1950s, many researchers have investigated the subject of garbage-collecting memory management. Baker [1] provided the foundation for incremental collectors in 1978. (Cohen [2] provides an excellent survey of work done before 1981.) The technology was extended to generational collectors by Lieberman and Hewitt [3] in 1983. Moon described the practical issues and experience of garbage collection in a large Lisp system [4] in 1984. The prime issue to be addressed in this article is an extension to prior work to improve locality of reference in a garbage-collected system by the application of an adaptive memory management algorithm.

2. BASIC STRUCTURE OF A COPYING GARBAGE COLLECTOR

The garbage-collection process starts with an atomic action called a *flip*. As a result of a flip, the address space of the system is logically divided into three disjoint space types, as shown in Figure 1. All of the objects to be tested for accessibility are classified as *FROM-SPACE*. All of the objects not included in *FROM-SPACE* that may contain pointers to objects located in *FROM-SPACE* are classified as *SCAVENGE-SPACE*. Any objects not included in FROM-SPACE that can be certified as *not* having any pointers to objects in FROM-SPACE are classified as *NEW-SPACE*. This subdivision of the address space is done entirely by assigning logical space type attributes to the various regions; no objects are copied to effect the division. In some systems, the initial size of NEW-SPACE after the flip is zero.

After a flip, each word of SCAVENGE-SPACE is examined to determine if it contains a pointer to an object located in FROM-SPACE. When a pointer to an object in FROM-SPACE is found, the object is copied to SCAVENGE-SPACE in front of the current scavange location such that the words of the copied object will also be examined in the future. The original pointer to the object in FROM-SPACE is changed to point to the new object location in SCAVENGE-SPACE. The original object location in FROM-SPACE is filled with forwarding pointers to the new object location in SCAVENGE-SPACE. If a second pointer to the same object is subsequently processed, we can simply

change the pointer to the new location. In any case, the word in SCAVENGE-SPACE does not contain a pointer to FROM-SPACE after it has been processed, so the classification of the word is changed to NEW-SPACE. The previously described process is called *scavenging*, and the systems element that is responsible for scavenging is called the *scavenger*.

Eventually, the scavenger converts every word of SCAVENGE-SPACE to NEW-SPACE, and then all of FROM-SPACE can be safely reclaimed because NEW-SPACE is guaranteed to have no pointers to FROM-SPACE.

3. BASIC STRUCTURE OF A TEMPORAL (GENERATIONAL) GARBAGE COLLECTOR [3]

Temporal garbage collection (TGC) is based on two heuristics:

- Young objects tend to become garbage quickly.
- The number of pointers from old objects to young objects is relatively small.

The first heuristic suggests that it would be smart to classify objects into age-oriented generations so that we can collect the generations with young objects much more often than the generations with old objects. The second heuristic suggests that we should remember the location of pointers to young objects so we can avoid scavenging the entire address space when collecting a young generation.

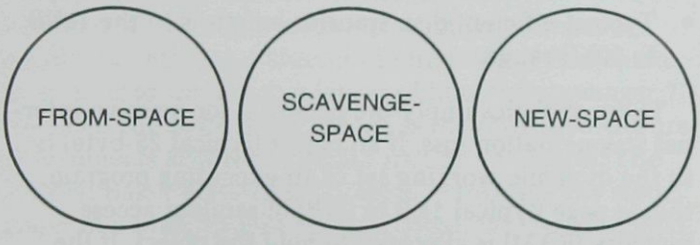

FIGURE 1. Space Types in a Copying Garbage Collector

These two objectives can be satisfied by dividing the address space into regions as represented by the circles in Figure 2. Four generations are supported, represented by the rows in Figure 2. The address space is also characterized by volatility, which is represented by diagonals in Figure 2. The volatility of a region specifies the youngest generation that can be pointed to by a pointer stored in the region. For example, the 3,3 region can contain only pointers to objects in generation 3 (the bottom row in Figure 2). On the other hand, the 3,0 region can obtain pointers to objects in any generation.

Each write operation is checked for consistency with the volatility constraints. The Explorer® workstation includes hardware support to do this consistency check in parallel with the write operation. When a volatility violation is detected, a trap to a volatility exception handler is generated. For example, an exception trap is generated if an attempt is made to store a pointer to a generation 0 object in the 3,3 region. The volatility exception handler must resolve such conflicts. One design choice would be to cause the volatility associated with the store address to be changed. In a demand paged virtual memory system, this design choice can be effected by marking the page as containing pointers to young objects [4].

Another design choice is to introduce a level of indirection. The volatility exception handler first allocates a single word of new storage in the region with generation equal to that of the fault store address and volatility equal to the generation of the object being pointed to. Then, the original pointer is stored in this newly allocated cell, and an indirection pointer to the newly allocated cell is stored in the original location. This is the approach taken in the TGC system. Using the volatility-violation example given above, a single word of storage is allocated in the 3,0 region, and the pointer to the generation 0 object is stored in this new cell. Then, an indirection pointer to the new cell is stored in the original location. This approach allows all of the pointers from old objects to young objects to be grouped together so that the scavenger can access them efficiently without accessing pages scattered about the address space. This approach also introduces a level of dynamic indirection not present in the page marking scheme. Performance measurements, however, indicate that this level of dynamic indirection typically uses less than 3 percent of the processor time—a good tradeoff against the reduction in disk-wait time obtained.

Figure 3 shows the functional usage of the regions. Most application objects of the system are located in the left column where the generation is equal to the volatility. These regions are much larger than the other regions and typically represent 95 percent of the total

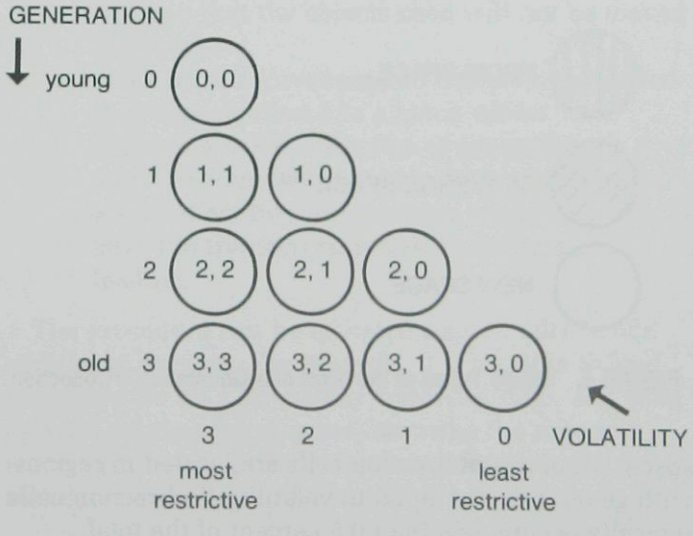

FIGURE 2. Address Space Divided by Generation and Volatility

Explorer and Explorer II are trademarks of Texas Instruments Incorporated.

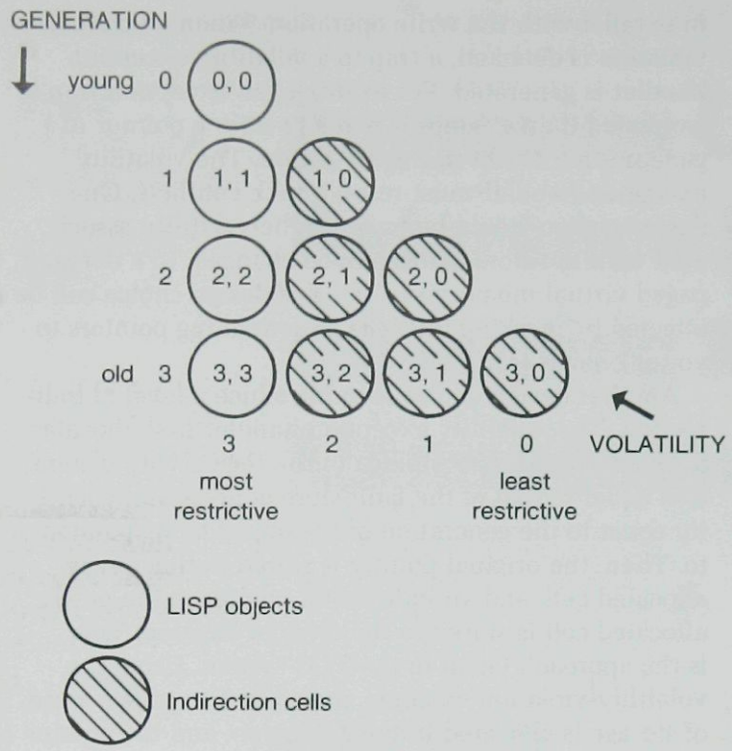

FIGURE 3. Functional Usage of Regions

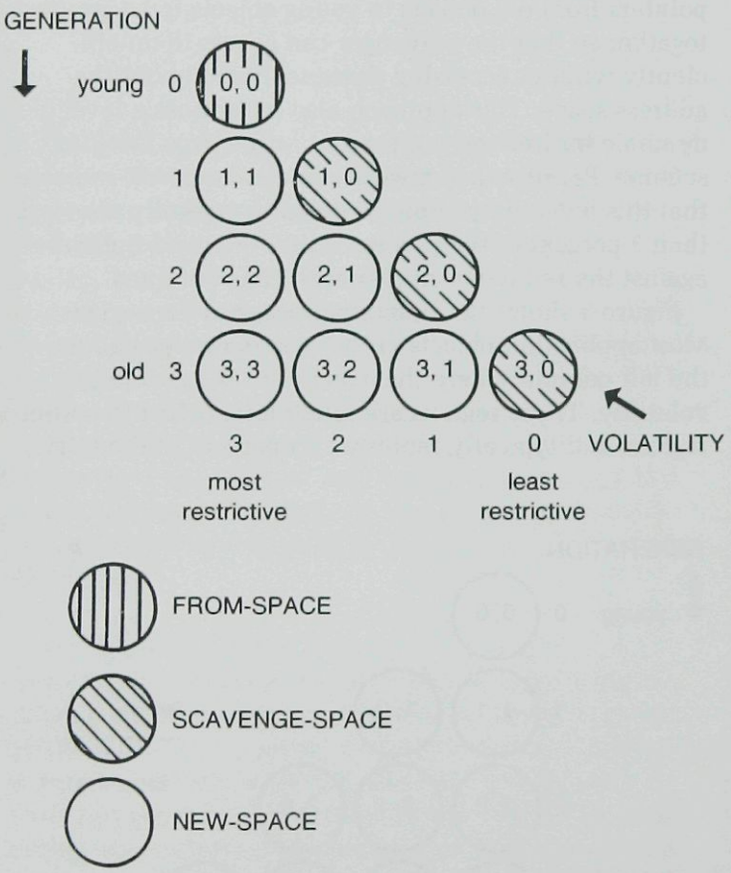

FIGURE 4. Space Types at the Start of a Generation 0 Collection

space. All of the indirection cells are located in regions with generation not equal to volatility (indirection cells typically occupy less than 0.5 percent of the total space). The process stacks of the system are located in the regions with volatility zero and typically represent 5 percent of the total space.

Figure 4 shows the configuration of the system at the start of a generation 0 collection. Note that most of the memory content (located in the left column) can be precertified as free of any pointers to objects located in FROM-SPACE. We need to scavenge only the volatility 0 regions. The objects that survive the generation 0 collection are promoted to generation 1.

Figure 5 shows the configuration of the system at the start of a generation 1 collection. Here the volume of information that must be scavenged has grown slightly, but a large part of the system can still be precertified as free of pointers to objects located in FROM-SPACE. When generation 1 is collected the indirection cells located in generation 1 also are converted to FROM-SPACE. This provides a natural solution to the question: How does an indirection cell ever go away? The processing of the indirection cells converted to FROM-SPACE effects a test for continued need. An indirection cell is regenerated if it is still needed and eliminated if it is not still needed. Often they are not still needed because the object that caused the original creation of the indirection cell has either become garbage or been promoted to generation 1. The objects that survive the generation 1 collection are promoted to generation 2.

4. LOCALITY OF REFERENCE

Consider the following statistics:

- The average atomic object size in a Lisp system is 25 bytes.
- Typical page size in demand paged virtual memory systems is in the 1KB to 8KB range.
- Typical efficient disk transfer length is in the 10KB to 30KB range.

These statistics imply the potential for massive internal fragmentation loss. If an object (typical 25-byte) is in the dynamic working set of an executing program, then a page (typical 1KB to 8KB) of random access memory (RAM) is allocated to hold the object. If the other objects located in the same page are not also in the dynamic working set, then we have done a very poor job of managing a relatively expensive resource. Also, if there is to be any hope for effective use of prepaging, we need locality of reference over a distance that spans several pages.

Past work has noted that locality of reference is one of the major reasons for choosing a copying garbage collector in a virtual memory system. Timeliness is also important, and the generational algorithm helps in this regard. However, even if the objects placed near a dynamically active object are not garbage, if they are not dynamically active at the same time, then we can have serious internal fragmentation loss. Since the garbage collector is going to be moving objects in the virtual address space anyway, it ought to be able to improve placement for locality of reference.

In summary, there is great potential for gain if we can arrange the placement of objects such that when an object is needed, it is very likely that other objects placed near it will also be needed.

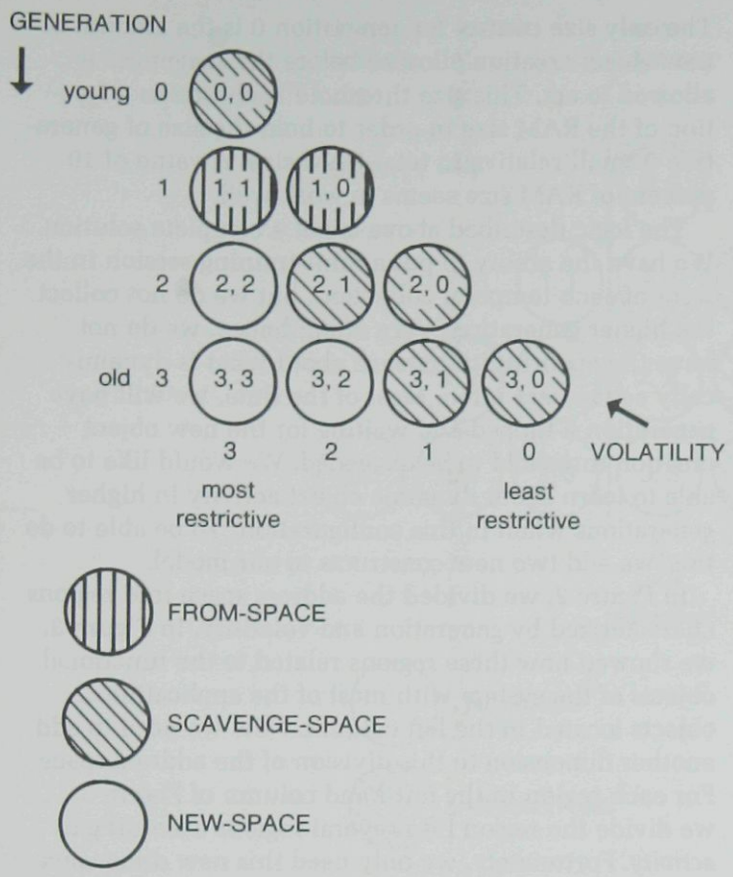

FIGURE 5. Space Types at the Start of a Generation 1 Collection

4.1 Static Graph Object Placement

How can we affect the object placement to improve locality of reference? One of the first things that comes to mind is to control the details of how the scavenger traverses the *static graph*, which is the transitive closure of all objects accessible from a root set of objects. The original Baker algorithm [1] tends to traverse the static graph in a breadth-first fashion. Moon [4] has described a modification that achieves an approximately depth-first graph traversal. TGC incorporates a stack in the scavenger to allow an object-oriented, depth-first, scavenging sequence. The results seem to indicate that the depth-first traversal of the static graph is superior to a breadth-first approach. However, the resulting gains obtained are disappointingly small (approximately 10 to 15 percent). The problem is that the static graph contains information about how objects *can be* accessed but not how objects *will be* accessed.

4.2 A Watershed Experiment

Consider the following simple experiment:

1. Flip the entire system to FROM-SPACE.
2. Inhibit the scavenger.
3. Use the system (read your mail, use the editor, compile a program, execute your favorite applications, and so on). Drive the system through the most frequent set of functions that you use. As you do this, on an object-by-object basis, the objects needed to support your activity will be copied from FROM-SPACE in the order that they are needed. The objects not needed to support your activity will stay in FROM-SPACE. Note that we are forcing an exposure of the objects that will be accessed, and this is the information that is not contained in the static graph alone.

After the system has been used for a while, we make two observations. Even after substantial usage, we find that the volume of information copied out of FROM-SPACE is very small relative to the total size of FROM-SPACE. On an Explorer workstation, this volume is about 4MB when FROM-SPACE is about 30MB. This is only 13 percent! Second, there is little paging activity. Now, one could interpret this result as indicative of a need to "get rid of the other stuff that I am not using." This could be a mistake. There is no evidence that you will not want to access this information in the future. For example, when you go to a university library, you do not access all the information in the library for a particular research project, but you know—intuitively—that the needs of your immediate project say little about the usefulness of the other information in the library.

4.3 Band Training

The results obtained from the preceding experiment have been formalized into a procedure for training a system before its image is saved on disk for future loading. The procedure consists of the following steps:

1. The user executes the Lisp function (START-TRAINING-SESSION). This function flips generation 3 to FROM-SPACE and inhibits the scavenger.
2. The user exercises the most frequently used systems functions. During this training session, the objects needed are copied from FROM-SPACE in the order they are needed.
3. When the user is finished with the training session, the function (END-TRAINING-SESSION-AND-DISK-SAVE) is executed. This function will:
   a. Make the objects copied from FROM-SPACE static so that the objects used will not be moved in the future.
   b. Activate the scavenger and complete the collection of generation 3 in a batch mode.
   c. Perform a batch collection of generations 0, 1, and 2 to eliminate the garbage generated as a side-effect from the training session.
   d. Save the trained image to disk for future loading.

The procedure can be repeated sequentially to add additional training. For example, it is possible to load new software on a previously trained system, and then execute another training session using the new software. This training results in the performance improvements which are described in Section 5.

4.4 Adaptive Training

The band training facilities described previously leave some serious questions unanswered:

- What about the data set created after a boot? How can one apply dynamic object placement to these objects?
- What if my (personal) usage pattern shifts from the usage pattern applied in generating a trained band?

Concern for these questions had led to the development of a way to integrate an automatic adaptive training facility into the TGC system. The system extracts dynamic access information to control object placement to improve locality of reference.

The systems element responsible for the evacuation of objects from FROM-SPACE in the Explorer workstation is a microcode-implemented routine called the *transporter*. The transporter can be invoked by two distinctly different mechanisms:

- The executing program, sometimes called the *mutator*, can make reference to an object located in FROM-SPACE, causing the transporter to be invoked in order to copy the object from FROM-SPACE. This form of transporter invocation implies a very positive assertion that the object involved is dynamically active.
- The scavenger may process a word that points to an object located in FROM-SPACE, causing the transporter to be invoked in order to effect the necessary evacuation. This form of transporter invocation does not imply dynamic activity. The object is not garbage, and it must be moved. However, in a somewhat more passive "condemnation by faint praise" sense, we have some reason to believe that the object is not dynamically active.

The preceding observation suggests that we should hold the scavenger inhibited for a while after a flip to allow the mutator the opportunity to cause dynamically active objects to be copied before the inactive objects are moved by the scavenger. This places a mini-training session on the front of each temporal collection cycle. The inhibition of the scavenger is done by allowing a threshold volume of new object allocation to occur after a flip before the scavenger is allowed to act. A subtle change in flip control logic is also used for generation 0. In past incremental collection control algorithms, the common sequence has been:

- Wait until some size threshold is exceeded, then atomically perform a flip, scavenge (incrementally) until complete, reclaim FROM-SPACE, Wait until some size threshold is exceeded, . . .

This control sequence has been changed to:

- Flip a generation greater than 0 if it has exceeded a size threshold, else flip generation 0 immediately, allow a volume of new object creation to exceed a size threshold before the scavenger is allowed to act, scavenge (incrementally) until complete, reclaim FROM-SPACE, Flip a generation greater than 0 . . .

In the second control sequence, some generation is flipped (usually generation 0) almost all of the time. The only size control for generation 0 is the amount of new object creation allowed before the scavenger is allowed to act. This size threshold is chosen as a fraction of the RAM size in order to hold the size of generation 0 small relative to total RAM size. A value of 10 percent of RAM size seems to work well.

The logic described above is not a complete solution. We have the ability to put a mini-training session in the front of each temporal collection, but we do not collect the higher generations very often; hence, we do not have the opportunity to learn about what is dynamically active very often. Most of the time, we will have generation 0 flipped and waiting for the new object creation threshold to be exceeded. We would like to be able to learn about dynamic object activity in higher generations when in this configuration. To be able to do this, we add two new constructs to our model.

In Figure 2, we divided the address space into regions characterized by generation and volatility. In Figure 3, we showed how these regions related to the functional objects of the system with most of the application objects located in the left column. Now we need to add another dimension to this division of the address space. For each region in the left-hand column of Figure 3, we divide the region into several regions according to *activity*. Fortunately, we only need this new dimension for the left-hand column, so we can fold these new regions back into our two-dimensional representation, as shown in Figure 6. We now have a data structure capable of representing dynamic object activity with four different grades ranging from 0 (most active) to 3 (least active).

In addition to the activity dimension described above, we define a new space type called *TRAIN-SPACE*. This new space type has access rules that are very similar to those that apply for FROM-SPACE:

- If the mutator attempts to access an object located in TRAIN-SPACE, the transporter will evacuate the object. This rule treats TRAIN-SPACE the same as FROM-SPACE.
- If the scavenger processes a pointer to an object located in TRAIN-SPACE, the transporter will not evacuate the object. This rule treats TRAIN-SPACE differently than FROM-SPACE.

Finally, we add two new rules for the transporter to follow:

- When the transporter is invoked to move an object for the mutator, the new copy is placed in a region with activity 0.
- When the transporter is invoked to move an object for the scavenger, the new copy is placed in a region with activity one greater than its current activity (or 3 if its current activity is 3).

Figure 7 shows these new transporter rules graphically. The objects evacuated from TRAIN-SPACE can be placed directly in NEW-SPACE since they cannot contain pointers to objects in FROM-SPACE. Only the

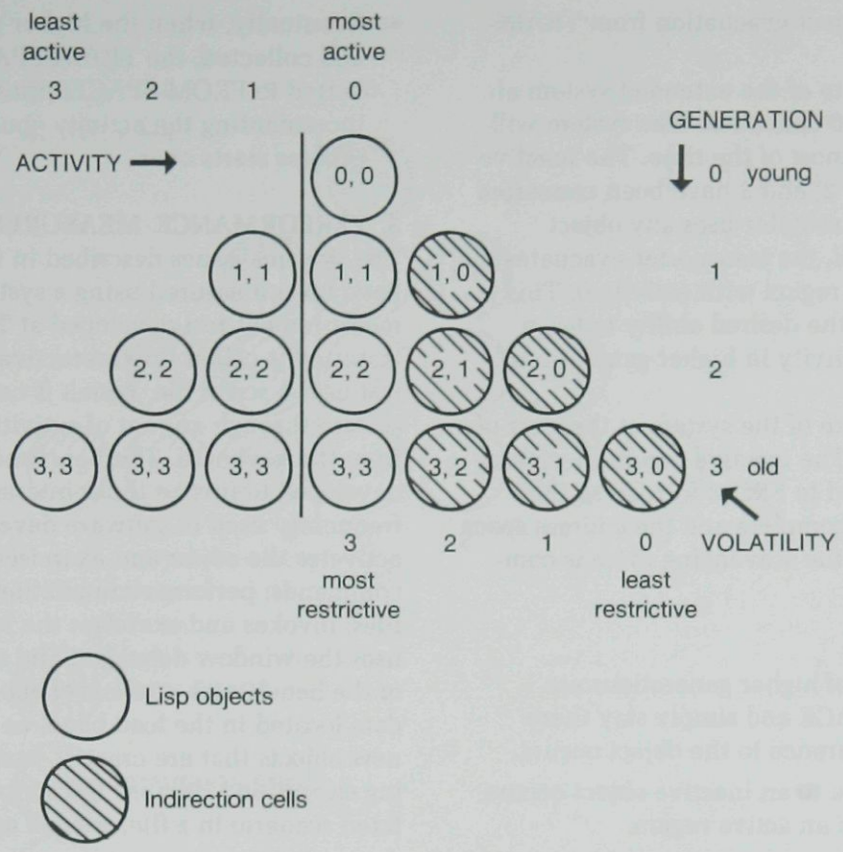

FIGURE 6. Functional Usage of Address Space also Divided by Activity

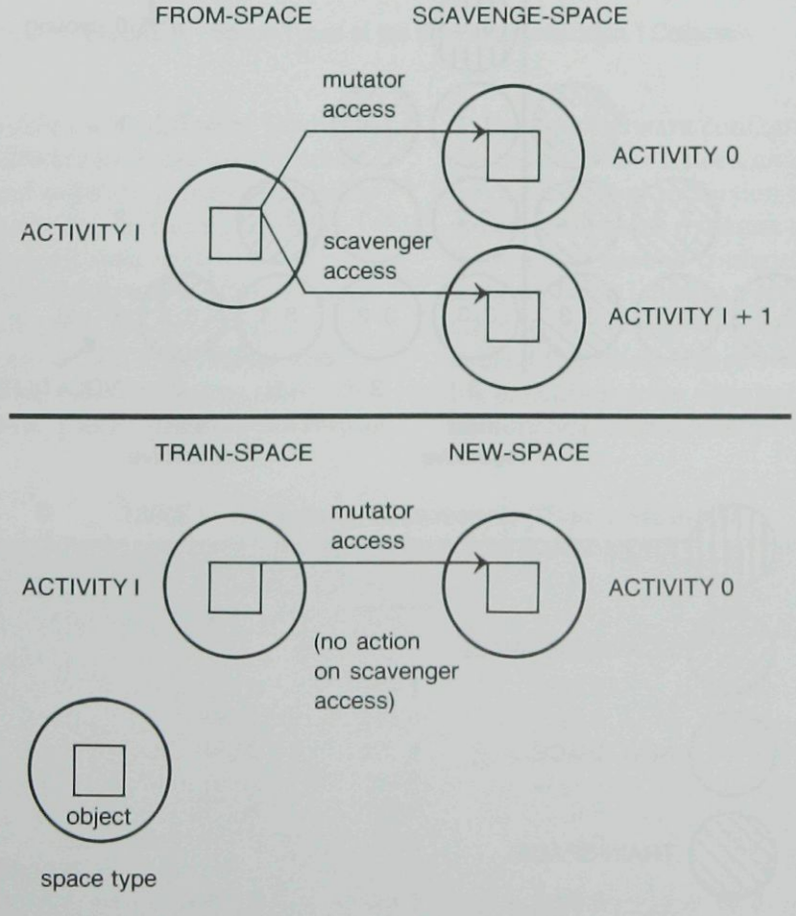

FIGURE 7. New Transporter Rules

mutator can cause an object evacuation from TRAIN-SPACE.

Figure 8 shows the state of the extended system at the start of a generation 0 collection. The system will be in this configuration most of the time. The inactive regions in generations 1, 2, and 3 have been converted to TRAIN-SPACE. If the mutator uses any object located in TRAIN-SPACE, the transporter evacuates the object to a NEW-SPACE region with activity 0. This object faulting provides the desired ability to learn about dynamic object activity in higher generations essentially all the time.

Figure 9 shows the state of the system at the start of a generation 1 collection. The inactive region in generation 1 has been converted to FROM-SPACE so that evacuation will now be complete and the address space will be reclaimed when the scavenging cycle is complete.

In summary:

- The inactive objects of higher generations are flipped to TRAIN-SPACE and simply stay there unless a dynamic reference to the object occurs.
- If a dynamic reference to an inactive object occurs, the object is copied to an active region.
- Eventually, when the higher generation is flipped and collected, the TRAIN-SPACE regions are converted to FROM-SPACE, prior training is retained by incrementing the activity counters, and the learning process starts over.

5. PERFORMANCE MEASUREMENTS

The systems issues described in the previous sections have been measured using a systems-level performance measurement test developed at Texas Instruments. The test used is called the Interactive Benchmark test. This test uses a script file, which is capable of driving the system through any set of activities that can be invoked from the keyboard. The benchmark script has been developed to invoke those interactive functions most frequently used in software development. The script activates the editor and exercises some of the editor commands; performs compilations of both buffers and files; invokes and exercises the inspector; activates and uses the window debugger; and so on. (The working set of the benchmark consists of about 4.5MB of code and data located in the load band, as well as about 27MB of new objects that are created, used, and discarded during execution.) Because the script is contained as a fixed scenario in a file, we can apply a nontrivial,

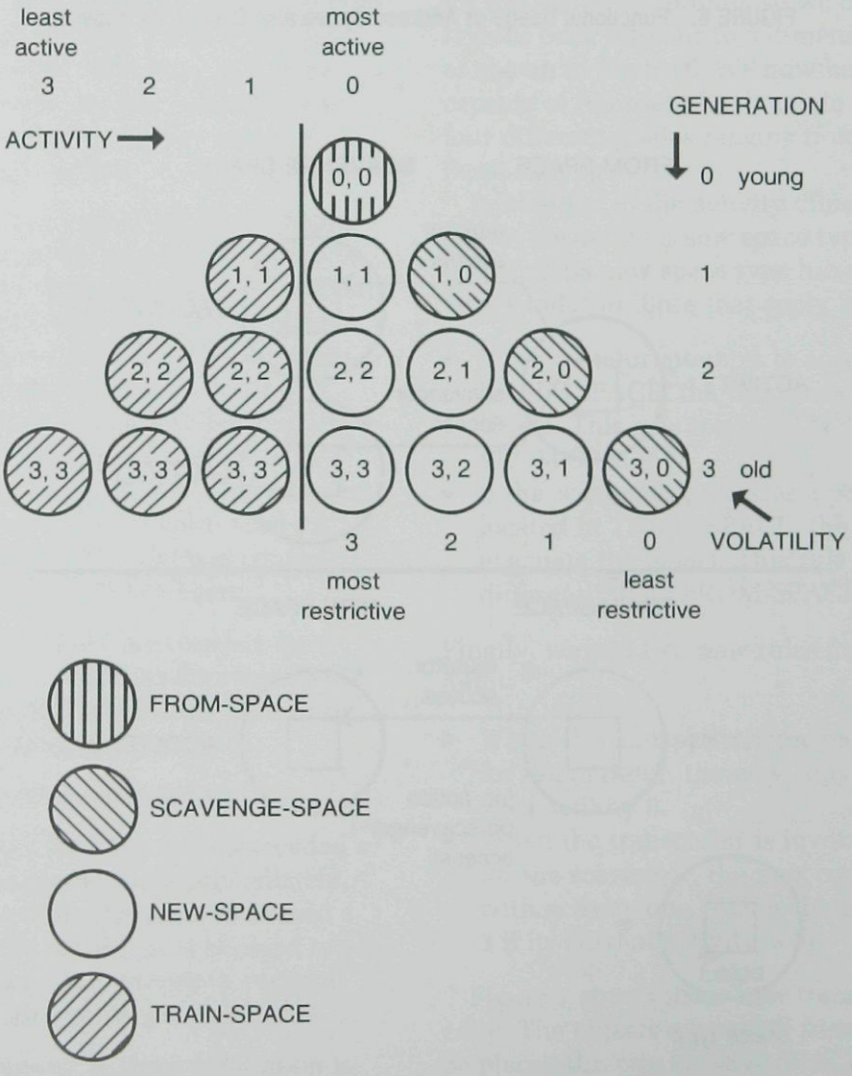

FIGURE 8. Space Types at the Start of a Generation 0 Collection

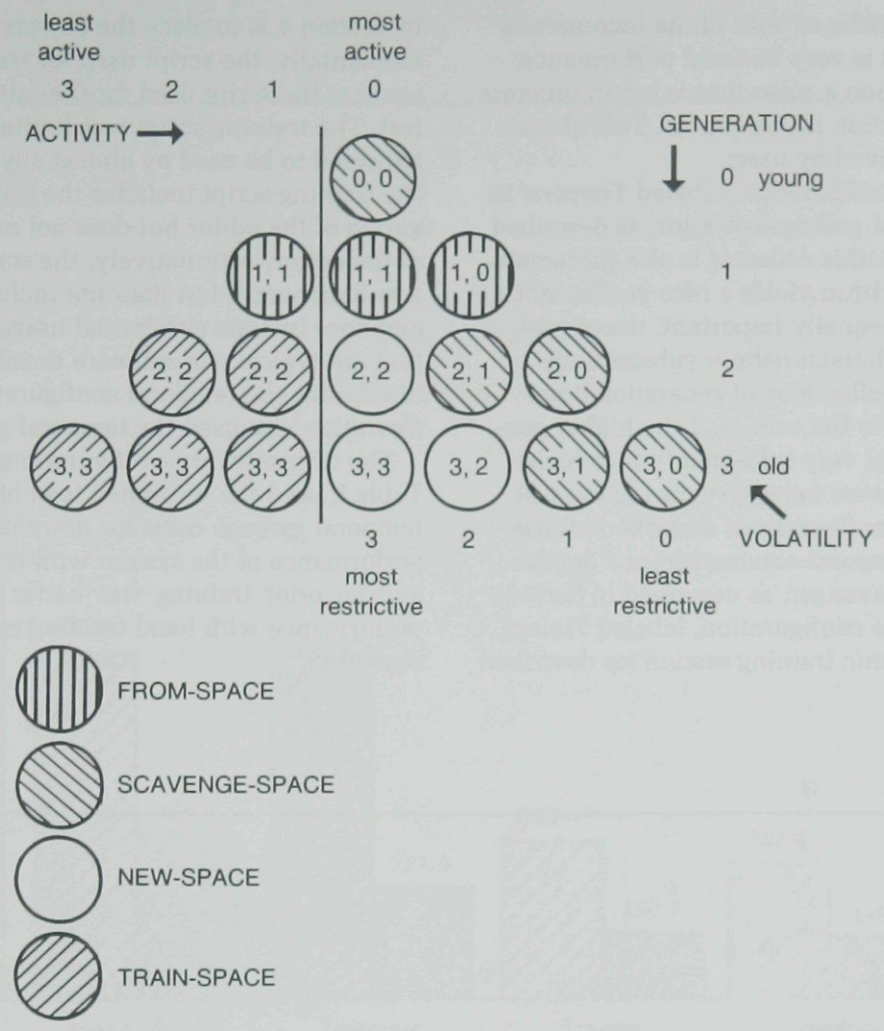

FIGURE 9. Space Types at the Start of a Generation 1 Collection

repeatable scenario to systems with different hardware and software. The test software includes no keystroke input time delays. The test software is instrumented to gather performance-related data, including the total time, CPU time, and page fault wait time.

Performance was measured using two different hardware configurations. In all cases, the hardware used was an Explorer II™ processor with two 140MB disk drives configured to provide 75MB of paging store. The RAM size was 4MB or 8MB. Table I presents the results.

The first software configuration, labeled *Incremental-dynamic* in Table I, used an incremental garbage collector, as described in Section 2 of this article. Almost all of the system was dynamic and subject to garbage collection. The second configuration, labeled *Incremental-static* in Table I, used an incremental collector also, but all of its initial system content was made static and not subject to garbage collection. The objects created during execution were dynamic and subject to garbage collection. Scavenging the entire system was still required.

TABLE I. Performance Measurements (Times in Seconds)

	Paging time		CPU time		Total time	
	4MB	8MB	4MB	8MB	4MB	8MB
Automatic Garbage Collection On						
Incremental-dynamic	1174.0	718.3	227.0	221.1	1401.0	939.4
Incremental-static	501.3	202.0	126.3	122.4	627.6	324.4
Temporal	482.2	121.4	110.7	100.0	592.9	221.4
Trained	200.0	58.2	104.4	98.5	304.4	156.7
Adaptive	122.6	39.2	119.8	108.9	242.4	148.1
Trained-adaptive	117.6	25.2	113.0	102.5	230.6	127.6
Automatic Garbage Collection Off						
Untrained band	381.9	113.2	98.8	92.6	480.7	205.8
Trained band	120.8	63.5	93.4	91.8	220.2	155.3

An important characteristic of both of the incremental software configurations is very bimodal performance: the systems run fast when a collection is not in progress and slow when a collection is in progress. This characteristic is not well received by users.

The third software configuration, labeled *Temporal* in Table I, used a temporal garbage collector, as described in Section 3. (Of course, this collector is also incremental.) The temporal algorithm yields a nice performance improvement. Perhaps equally important, the severe bimodal performance characteristic is substantially ameliorated also. The collections of generation 0 are essentially transparent to the user, and the higher generation collections occur very infrequently. No higher generation collections were necessary during the test.

The three software configurations described all use a static graph object placement obtained from a depth-first, object-oriented scavenger, as described in Section 4.1. The fourth software configuration, labeled *Trained* in Table I, used a dynamic training session (as described in Section 4.3) to place the objects in the load band. Incidentally, the script used for training was not the same as the script used for the Interactive Benchmark test. The training script was limited to those operations expected to be used by almost any user. For example, the training script includes the boot sequence and activation of the editor but does not make substantial use of the editor. Alternatively, the script for the Interactive Benchmark test does not include the boot sequence and does include substantial usage of system elements that are typical of a software development environment. Like the temporal configuration, the trained configuration also used the temporal garbage collector.

The fifth software configuration, labeled *Adaptive* in Table I, used the untrained load band and the adaptive temporal garbage collector described in Section 4.4. The performance of the system with the adaptive algorithm and no prior training was better than the system performance with band training and no adaptive algorithm.

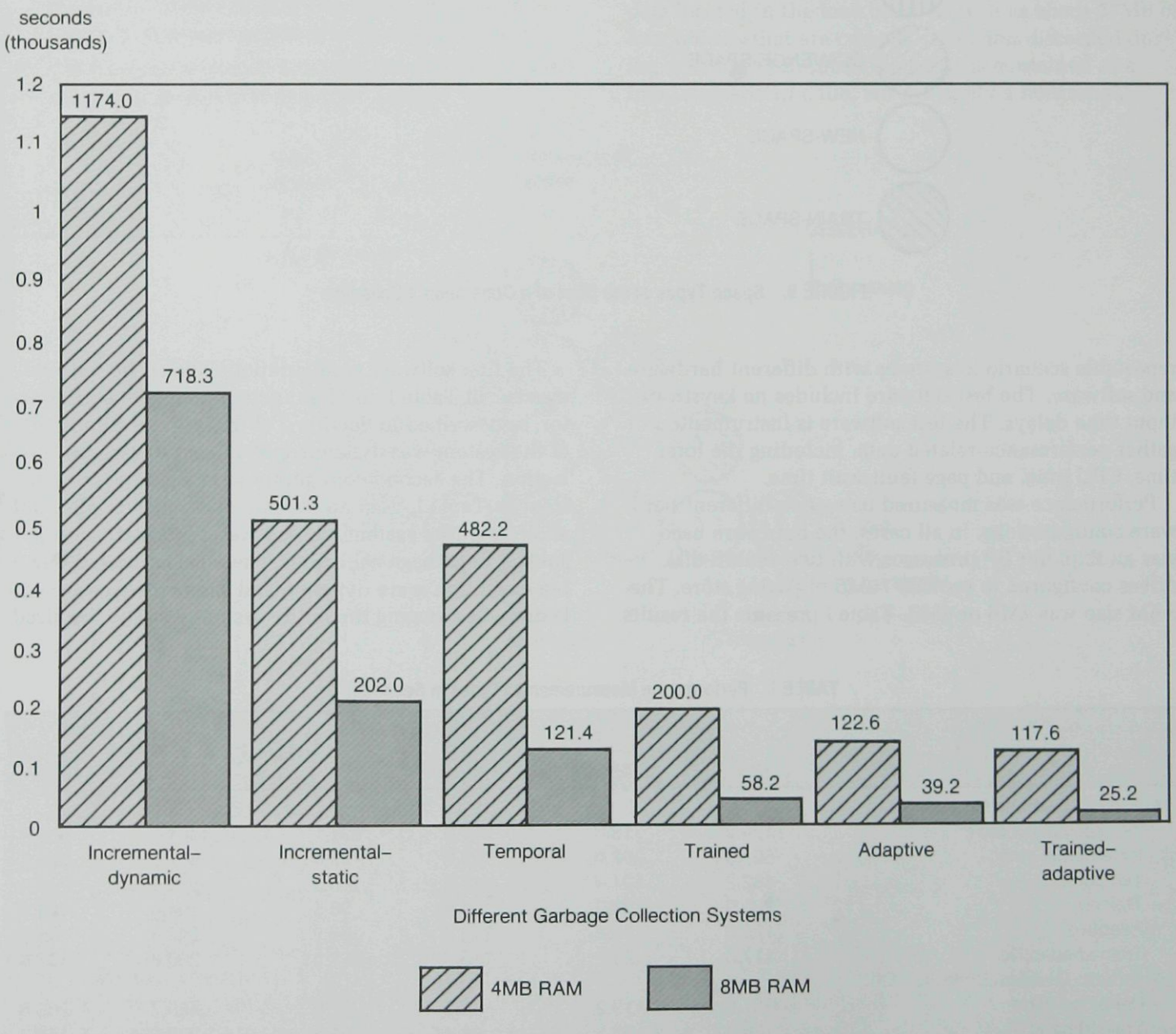

FIGURE 10. Paging Time, Automatic GC On

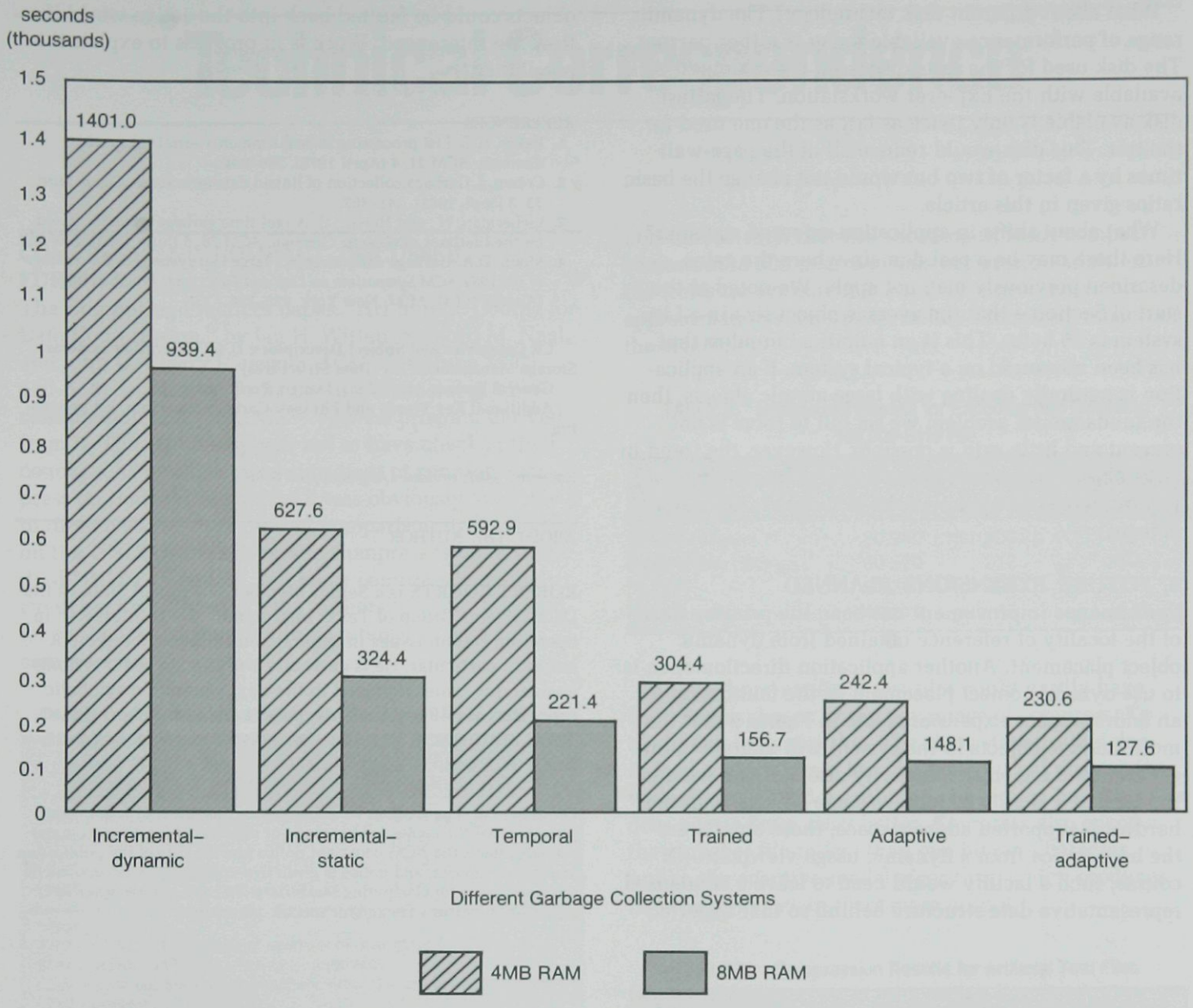

FIGURE 11. Total Time, Automatic GC On

The sixth software configuration, labeled *Trained-adaptive* in Table I, used the trained load band and the adaptive temporal garbage collector. This system provides the best performance for this test. However, in the presence of the adaptive training facility, the need for band training is reduced to an issue of first impressions. A system without any band training will be slow initially after boot, but band training will make little difference (and could result in degradation) after substantial usage.

Each of the two load bands was also measured with automatic garbage collection turned off. These results are also given in Table I. In terms of total execution time, the adaptive system—with 8MB RAM and garbage collection turned on—is faster than the trained band with garbage collection turned off.

Figures 10 and 11 graphically show the performance data for paging time and total time, respectively. If we compare the paging time of the temporal system to the paging time of the trained-adaptive system, we find that the dynamic object placement facilities reduce the paging time by a factor of 4.1X at 4MB and a factor of 4.8X at 8MB. On the other hand, if we compare the total time of the 8MB temporal system to the total time of the 4MB trained-adaptive system, we see that the performance is about the same. Thus, the impact of the dynamic object placement facilities is nearly the same as doubling the RAM size.

When do the gains described above apply and when do they not apply? Clearly these gains do not apply to a system configured with a RAM size that is large relative to the application working set. For example, a 32MB RAM would be large for the working set of the test described above, so paging would be minimal and no gain is possible. However, few users can afford to greatly oversize RAM. In the future, it is very likely that the cost of RAM will remain a significant part of total system cost.

What about different disk technology? The dynamic range of performance available today is rather narrow. The disk used for the tests discussed is the slowest disk available with the Explorer workstation. The fastest disk available is only twice as fast as the one used for the test. This disk would reduce all of the page-wait times by a factor of two but would not change the basic ratios given in this article.

What about shifts in application-oriented statistics? Here there may be a real domain where the gains described previously may not apply. We noted at the start of Section 4 that the average object size in a Lisp system is 25 bytes. This is an empirical number that has been measured on a typical system. If an application is naturally dealing with large atomic objects, then the fundamental problem we set out to solve is not present and little gain is possible. However, the trend in large object-oriented systems seems to be moving toward data structures with large numbers of small objects.

6. FUTURE EXTENSIONS PLANNED

Performance improvement has been the primary focus of the locality of reference obtained from dynamic object placement. Another application direction open is to use dynamic object placement as the foundation for an address space expansion. Note in Figure 9 that the most inactive objects in the system will migrate to the generation 3 region with activity also 3. If one were to implement a facility to remove some objects from the hardware-supported address space, these objects are the best choice from a dynamic usage viewpoint. Of course, such a facility would need to leave a condensed representative data structure behind so that exported objects could be faulted back into the active world if they are referenced. Work is in progress to exploit this possibility.

REFERENCES

1. Baker, H.G. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (April 1978), 280–294.
2. Cohen, J. Garbage collection of linked data structures. *Comp. Surv., 13*, 3 (Sept. 1981), 341–367.
3. Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM 26*, 6 (June 1983), 419–429.
4. Moon, D.A. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming* (August 1984). ACM, New York, 235–246.

CR Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; E.1 [**Data Structures**]
General Terms: Algorithms, Design, Performance
Additional Key Words and Phrases: Garbage collection, list processing

Received 9/87; revised 1/88; accepted 6/88

ABOUT THE AUTHOR:

ROBERT COURTS is a Senior Member, Technical Staff, in the Data Systems Group of Texas Instruments. He is interested in operating system issues in general and storage management issues in particular. He is currently working on some systems engineering issues involved in the next generation symbolic processing products. Author's present address: Robert Courts, Texas Instruments, Inc., Data Systems Group, P.O. Box 2909, MS 2201, Austin, TX 78769-2909.