Demonic Memory for Process Histories

Paul R. Wilson and Thomas G. Moher
Human-Computer Interaction Laboratory
Dept. of Electrical Engineering and Computer Science
University of Illinois at Chicago
Box 4348 (M/C 154), Chicago, IL 60680

1. INTRODUCTION

It is often useful to be able to access past states of a process. In debugging, for example, we may want to see previous states of a program's execution in order to home in on the bug that led to an observable error. For fault tolerance, it may be necessary to revert to a previous state to recover from an error; for optimistic concurrency control, similar recovery may be necessary in the face of a bad guess about possible concurrency. In a programming language, "undo" and "lookahead" constructs may allow the convenient expression of many algorithms, while in a general computing environment, the same kinds of operations are desirable for dealing with persistent data (e.g., files).

Demonic memory is intended as an acceptably efficient recovery mechanism that will allow fast access to data at any time scale from a fraction of a second to months or years. For example, in a debugging session, it should take no more than a few seconds to look a "back in time" a few minutes. It should also be possible to recall within a few minutes any past state of data, even if that state was modified months or years previously. This includes the ability to recall and capture transitory versions in the editing of a text or design document, or intermediate steps in an exploratory data analysis. Ideally, the programmer/user should never have to explicitly save data for later use; conceptually, all states of the system should be saved indefinitely.

The goal of our research is to show that it is possible to efficiently implement such a general recovery facility, and to develop an abstraction that allows it to be used effectively. If we are successful at achieving this vision of generality, it presents the possibility of a very powerful monolingual system in which a few simple operations apply at different levels with identical semantics [HeKl85]. This could be particularly appropriate for an environment in which many computations are interactively "steered" rather than straightforwardly programmed, as in scientific visualization, and for integrated systems in which the user/programmer distinction is blurred.

(Providing such a powerful memory facility will put a burden on the user interface of a system, in order that the user could use it without getting lost in uninteresting old data. We feel the user interface is the best place for such concerns, however; it is better to have too much data available than too little, and for system designers to choose features rather than having them dictated by efficiency considerations.)

If we are less successful in achieving generality and efficiency, we expect that specialized variants of demonic memory will be sufficient for more specialized domains, such our own work in reversible debuggers.

2. OVERVIEW

Demonic memory is a form of reconstructive memory for process histories. As a process executes, its states are regularly checkpointed, generating a history of the process at low time resolution. Following the initial generation, any prior state of the process can be reconstructed by starting from a checkpointed state and re-executing the process up through the desired state, thereby exploiting the redundancy between the states of a process and the description of that process (i.e., a computer program).

The reconstruction of states is automatic and transparent. The history of a process may be examined as though it were a large two-dimensional array, or *address space-time*, with a normal address space as one axis and steps of process time as the other. An attempt to examine a state that is not physically stored triggers a "demon" which reconstructs that memory state before access is allowed.

Regeneration requires an exact description of the original execution of the process. If the original process execution depends on non-deterministic events (e.g., user input), these events are recorded in an *exception list*, and are replayed at the proper points during re-execution.

While more efficient than explicitly storing all state changes, such a checkpointing system is still prohibitively expensive for many applications; each copy (or *snapshot*) of the system's state may be very large, and many snapshots may be required. Demonic memory saves both space and time by using a *virtual copy* mechanism. (Virtual copies share unchanging data with the objects that they are copies of, only storing differences from a prototype or original [MiBK86].) In demonic memory, the snapshot at each checkpoint is a virtual copy of the preceding checkpoint's snapshot. Hence it is called a *virtual snapshot*. In order to make the virtual snapshot mechanism efficient, state information is initially saved in relatively large units of space and time, on the order of pages and seconds, with single-word/single-step regeneration undertaken only as needed. This permits the costs of indexing and lookup operations to be amortized over many locations.

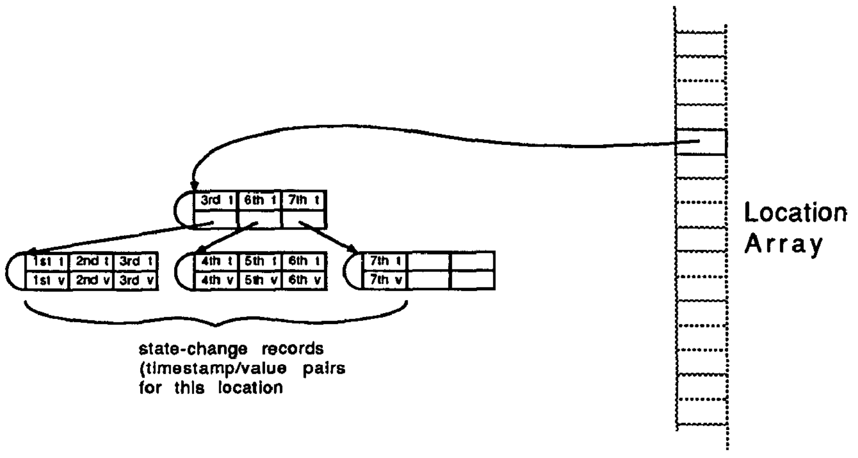


FIGURE 1a. Implementation of a location's history stack.

Each element of the location array holds a pointer to the root node of a tree. The leaf nodes (i.e., the bottom row) hold state-change records which comprise the stack of values taken on by the location. The non-leaf nodes form a tree-structured index into this stack.

Since state-change records are inserted in timestamp order, the stack grows monotonically to the right as process history is recorded.

Each node contains a pointer back to its parent node, and the root node always keeps a reference to the most recently accessed element of the tree (not shown.) These features facilitate searches of the tree. They especially facilitate successive insertions, since the last-accessed reference generally acts as a top-of-stack pointer.

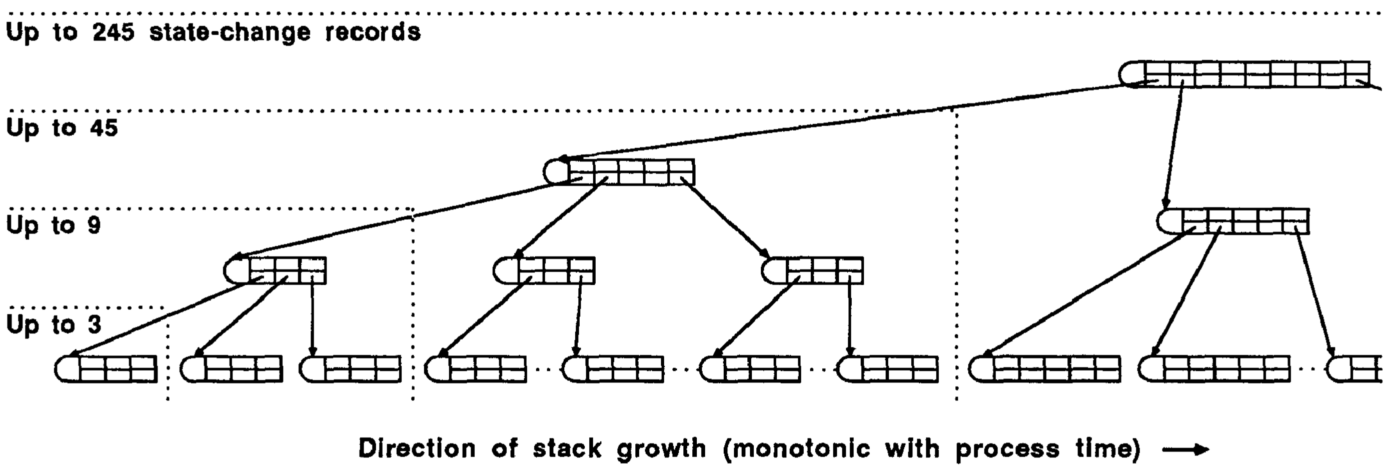
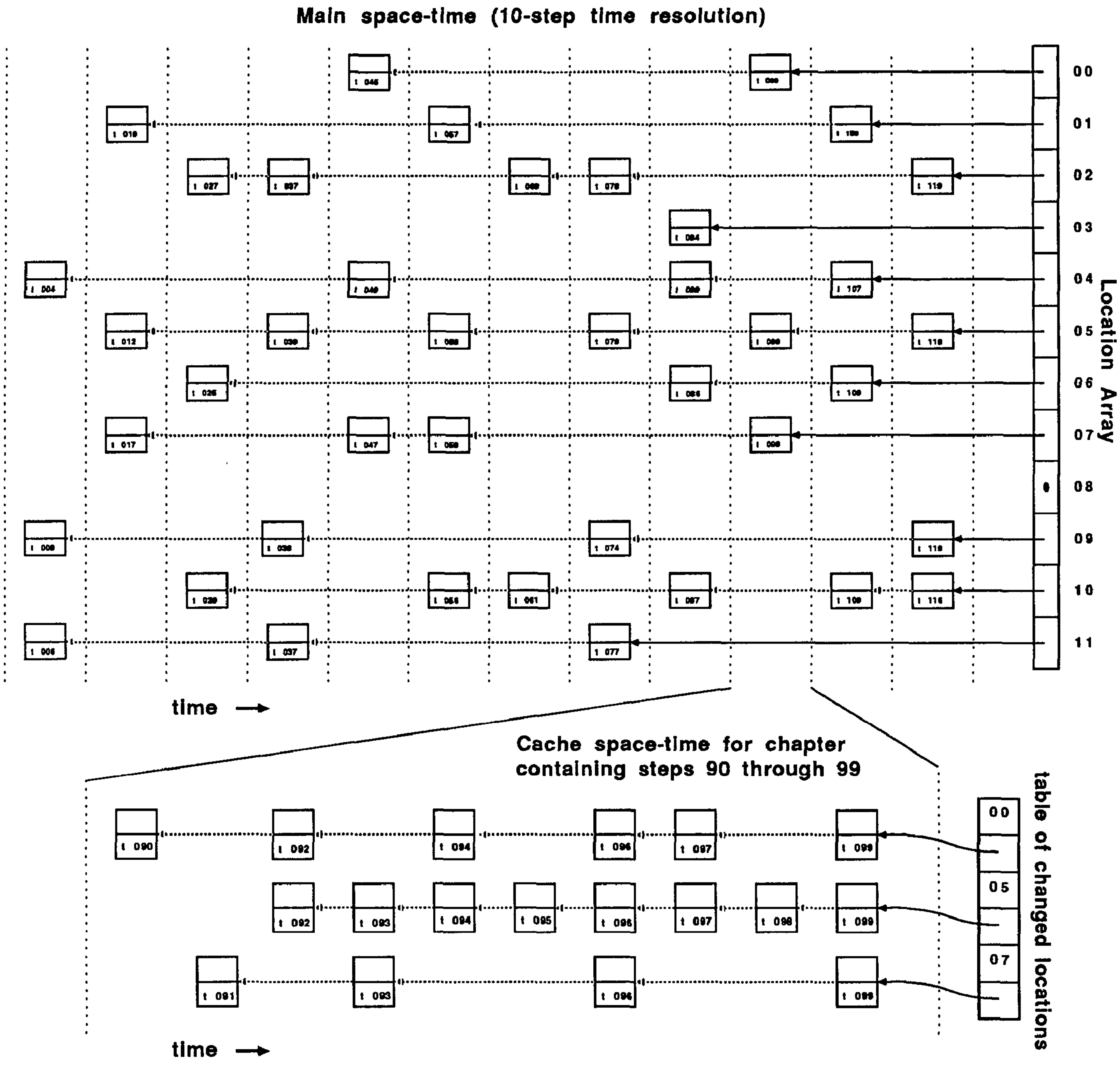

Up to 245 state-change records
Up to 45
Up to 9
Up to 3
Direction of stack growth (monotonic with process time)

FIGURE 1b. Growth of a location's history stack

History stack/tree growth is extremely regular, due to the constraint that records are inserted in key (timestamp) order. All trees with the same number of elements have exactly the same shape.

When a tree of a given number of levels is full, the next insertion causes a new, higher-level root node to be generated. The full tree then becomes the leftmost child of the new root node.

Because of this regularity, and with the help of backpointers and the last-accesed-reference cache, insertion and search time increase slowly with tree size.



Main space-time (10-step time resolution)
Location Array
00
01
02
03
04
05
06
07
08
09
10
11
time →
Cache space-time for chapter containing steps 90 through 99
table of changed locations
00
05
07
t 090
t 091
t 092
t 093
t 094
t 095
t 096
t 097
t 098
t 099
time →

FIGURE 2. A space-time implementation with multiple levels of time resolution.

A space-time is initially generated at a lowered level of time resolution. For each chapter, one state-change record is generated for each changed location. Subsequent changes to a location within a chapter cause that state-change record to be overwritten. At the end of the chapter it therefore contains the last value for that location within that chapter. In order to examine states within a chapter, a cache space-time must be generated, re-executing the process to generate state-change records for the chapter at single-step time resolution.

(Note that the location history stacks, suggested here by greyed pointers, are really tree-structured, as in Figure 1). Though in this example chapters are 10 steps long, they are actually thousands of times longer; there are also more levels of resolution than the two shown here.

One of the requirements of such a scheme is that there be considerable locality of state changes. If only a few locations change in many pages, much needless copying, indexing, and lookup of unaltered data will still be required. Thus we require a memory management scheme that tends to group state changes into a small subset of the pages of memory, within a given checkpointed interval. Coordination with garbage collection also allows garbage data to be omitted from checkpoints, a critical factor in demonic memory's efficiency. For these reasons we have devised a special-purpose garbage collector which operates in concert with demonic memory.

3. DEMONIC MEMORY BASICS

Demonic memory implements a *space-time abstraction.* It is logically a two-dimensional array, indexed by memory location (we assume word addressing here for simplicity) and process step. A step may be any action defined as atomic by the process interpreter that runs in demonic memory. ("Process interpreter" is used here to mean the process whose execution history is being recorded — it may be a compiled program as well as an interpreted one. The idea is that it interprets a process description and generates state changes to be recorded.)

3.1 Array-of-stacks implementation

If the logical array were implemented as a straightforward physical array, the time and space costs would be enormous. At each step it would be necessary to copy each state (array column) into the next column of the array, except for the changed elements.

The first step in making the abstract array efficient is to implement each row of the array as a *history stack*, indexed by time. An address space, then, is a single array, with each element representing the history of changes to a single memory location. The tops of the stacks act as a flat address space from the point of view of the history-generating process interpreter; it sees only the latest values for each location.

Each time a new value is stored into a location (pushed onto the location's history stack), it is paired with a timestamp indicating which step of the process generated it. In order to find the value of location **x** at time step **t**, we simply use **x** as an index into the location array and search the stack stored there for the most recent change prior to **t**.

Figure 1 shows an efficient way of implementing history stacks. The state-change records (timestamp/value pairs) are stored in the terminal nodes of a multi-way search tree. Since records are always inserted in key (timestamp) order, the tree grows very smoothly and simply; all trees with a given number of elements have exactly the same shape. When a tree of a given depth is full, it becomes the left subtree of a newly generated root node.

3.2 Multiple levels of time resolution

While the constraints on tree growth make access to state-change records locally efficient, the simple array-of-stacks implementation remains inefficient in a broader sense, since it stores every state change. Efficiency can be vastly improved by saving only occasional (checkpointed) states; intervening states can be reconstructed by re-running the process from the previous checkpoint.

Demonic memory implements a hierarchy of time granularities, storing only a subset of history at full resolution (single steps). Process history is divided into groups of consecutive steps called ***chapters***. A state-change record is created the first time a location is modified in a chapter, while subsequent changes within the chapter cause this information to be overwritten rather than pushed onto the history stack (see Figure 2). A state-change record is created the first time a location is changed within a chapter, but it ends up holding the last value, since all intervening values are overwritten; the process of overwriting successive changes amounts to checkpointing "on the fly."

In order to examine states within a chapter at single-step resolution, a *cache space-time* is generated on demand. The cache space-time holds the values stored into locations during the chapter. It is similar to the space-time just described, except that a hash table is used to implement the (sparse) location array. Regenerating requires memory fetches by the process interpreter, which keeps track of its "current" time (the step it has reached in regenerating history). A fetch must first look in the cache space-time, to see if the "latest" value for the location was generated within the chapter. If no value for the location is found in the cache space-time, the main space-time is searched for the last value generated prior to the chapter.

This arrangement is not limited to two levels. In the system we are implementing, the main space-time has "superchapters" many thousands of steps long. A few of these are fleshed out at a chapter-by-chapter level of resolution. At a given time, a few of these, in turn, are available at single-step resolution. This is analogous to keeping some pages of memory in RAM for high-speed access, but paging the rest out to disk, with "chapter faults" causing "chaptering" (regeneration) in much the same way that page faults cause paging in virtual memory.

3.3 Multiple levels of space resolution

The implementation described so far will still be rather slow. Stack operations are costly, and must be done often — on a per-location basis. If there is sufficient locality of state changes, the average cost of these operations may be reduced considerably by operating on memory a page at a time.

Rather than having a location array with a history stack for every location's values, a space-time can have a "page array" holding a history stack for each (logical) page. In these history stacks, a state-change record contains a reference to a physical page (RAM or disk), which holds values for an entire pageful of locations. Generation and regeneration of space-time are analogous to the methods described above

3.4 Using address maps to cache page lookups

Performance may be further improved by caching page versions using standard virtual memory hardware (and without interfering with its usual use). If the operating system allows manipulation of address mappings, all of the relevant page versions may be mapped into the address map used by an inspecting or history-generating process (see Figure 3, in which address maps are simply represented as raw page tables). The process then sees a normal, flat address space.

The virtual memory manager assists in other ways, as well. Write-protection is used to ensure that a new version of a page is generated the first time the page is modified within a chapter. At the beginning of each chapter, all pages are marked read-only; an

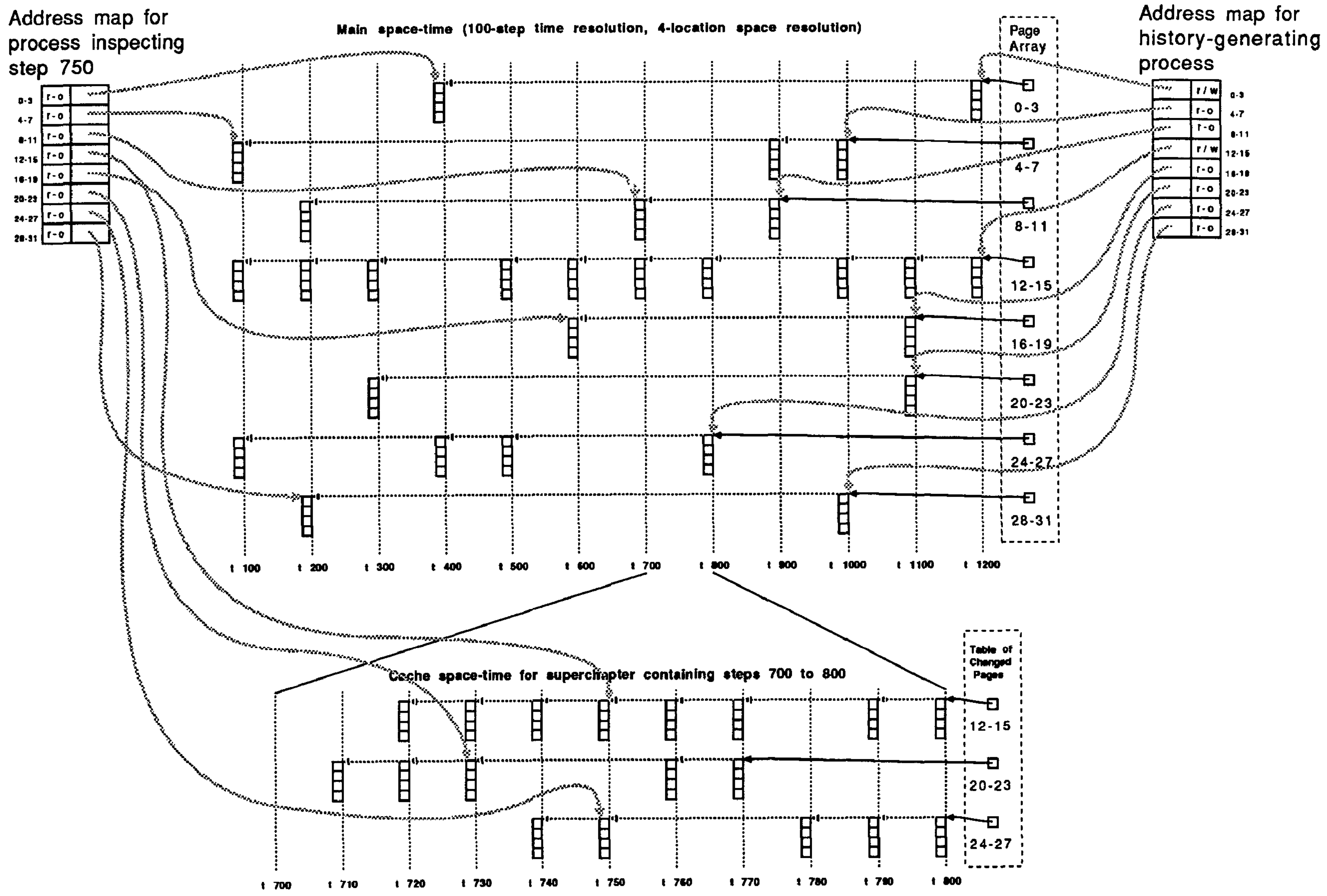

Address map for process inspecting step 750
Main space-time (100-step time resolution, 4-location space resolution)
Page Array
Address map for history-generating process
0-3
4-7
8-11
12-15
16-19
20-23
24-27
28-31
r-o
r/w
t 100
t 200
t 300
t 400
t 500
t 600
t 700
t 800
t 900
t 1000
t 1100
t 1200
Cache space-time for superchapter containing steps 700 to 800
Table of Changed Pages
t 700
t 710
t 720
t 730
t 740
t 750
t 760
t 770
t 780
t 790
t 800

FIGURE 3. Storing history at pagewise space resolution, using virtual memory address mappings as lookup caches

The address map used by the history-generating process (upper right) contains references to the most recent versions of all pages (i.e., the "tops" of all logical pages' history stacks). Thus the history-generating process sees its "current time" as a normal (flat) address space, and most memory references occur at normal hardware address-translation speeds. Pages generated during previous chapters are marked read-only; any attempt to modify them causes a trap that generates a new version. The address map used by the inspecting process (upper left) allows that process to see an old state as a flat address space, as well.

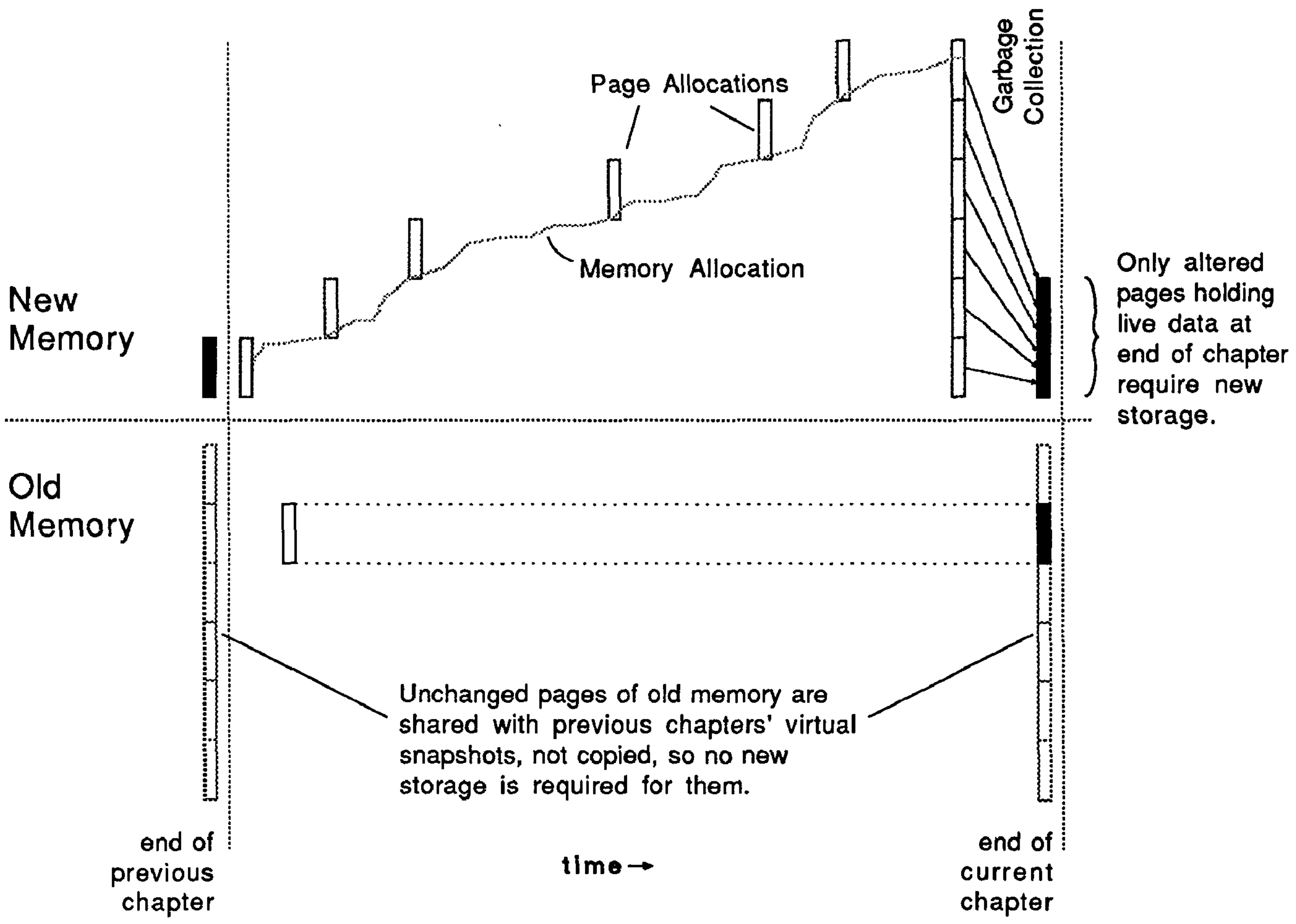

Garbage Collection
Page Allocations
Memory Allocation
New Memory
Only altered pages holding live data at end of chapter require new storage.
Old Memory
Unchanged pages of old memory are shared with previous chapters' virtual snapshots, not copied, so no new storage is required for them.
end of previous chapter
time →
end of current chapter

FIGURE 4. Only live-data pages must be recorded in virtual snapshots.

Data that become garbage by the end of the chapter needn't be stored; since subsequent states can't reference them, they may be left out of the virtual snapshot. If an attempt is made to access a state within a chapter (e.g., between checkpoints), the process is replayed to generate records of the intervening states. This will automatically regenerate any discarded data before it is accessed.

attempt to write to a page causes a copy to be made and pushed onto the history stack. It is then marked read/write until the end of the chapter.

Lookup of page versions may be done lazily. Parts of the flat address space may go unmapped initially; an unmapped-memory fault traps to a routine that looks up the correct page version and maps it into the address map for the process. (Copying may be used instead of remapping, if the operating system doesn't support remapping efficiently. Write protection can play the role of unmapped memory, with the write-exception handler may be conditionalized to do double duty. This would incur some extra cost in actual RAM and disk, however.)

These hardware-controlled trap routines act as demons that incur almost no extra overhead except when they are invoked. In addition, they exploit standard hardware without interfering with its standard uses. Most memory accesses in demonic memory can therefore occur as normal memory operations at normal (hardware) speeds.

4. HEAP MANAGEMENT

The efficiency of this page-based implementation depends critically on the locality of the state changes made by the history-generating process. In the worst case, the process could change a single location in a different page at every step in a chapter. Demonic memory would then make new versions of each of those entire pages, one per step.

Few processes exhibit such catastrophic tendencies; much activity in many languages occurs within a stack of activation records. In any reasonably short period of time, the stack usually varies in height within a rather constrained range. Heap data pose a more difficult problem, however. A classical stop-and-collect garbage collector approximates the worst case for demonic memory, since it uses a large amount of memory before reclaiming any.

Generation garbage collection [LiHe83] limits the scope of these problems. It is designed to reduce memory fragmentation (reducing storage requirements and virtual memory paging). Memory is divided into several regions, allowing more recently-allocated data to be garbage-collected independently of the older data. Since recently-allocated data are likely to become garbage quickly, and since long-lived data are likely to live still longer, this concentrates activity where it is most likely to reclaim significant storage.

For demonic memory purposes, the critical kind of program locality is the number of changed pages within a unit of time that must be checkpointed. Since garbage collecting a generation moves all of the live data in a copying/compacting operation, the pages that live data are moved to are necessarily altered in the process. The pages that end up holding these live data must therefore be stored by demonic memory.

It is important to note that pages that contain only garbage at the end of a chapter need not be saved in the chapter's virtual snapshot; if they hold no live data, there can be no pointers into them in pages shared by subsequent chapters' virtual snapshots. Any attempt to access states *within* the chapter will cause the data to be regenerated before they are accessed. (See Figure 4.)

Our garbage collector [Wils88b, WiMo89b] is most similar to Ungar's *Generation Scavenging* algorithm [Unga84, Unga86]. All objects are created in new memory. If an object survives a number of scavenges (collections) of the new region, it is copied out of that region into an older region that is scavenged less often. If there are more than two generations (as in our system), the same phenomenon repeats at the next level, at a much lower frequency.

Garbage collection and checkpointing may be coordinated to improve efficiency. Most higher-resolution cache space-times are quickly discarded, increasing the time granularity of stored history. Multi-generation garbage collection will occur at large-granularity time-resolution boundaries (e.g., ends of super-superchapters). Once the higher-resolution (e.g., chapter-by-chapter) information is discarded, memory requirements drop dramatically. There are two main reasons for this. First, longer duration snapshots contain at most one version of each page that changed within that interval, rather than one for each of the shorter subintervals. Second, many of those changed pages will hold only garbage at the end of the longer duration, and their contents need not actually be stored once the higher-resolution information is discarded. Thus old memory at low resolution takes comparatively little storage. (See Figure 5.)

5. SINGLE-STEP REGENERATION

Regenerating history at single-step time resolution poses special problems, especially when checkpointing on a paged basis. Using the same cache space-time mechanism used for lower-resolution generation will require copying whole pages at steps where one or a few locations change. A hash table of per-location history stacks could be used, as described in section 3; this would be very slow, however, since hashing operations are expensive, and it would defeat the use of page tables for caching version lookups.

Our approach is to use a per-page table, as when regenerating at single-chapter resolution. Only a single page version is created for each page changed in the chapter; each element of the changed pages table holds a single version, not a stack of versions. For those locations that only take on one value during the chapter, the value is simply stored in the appropriate location in the appropriate page. For any location that changes more than once, a forwarding pointer to a history stack for that location is installed. Thus instead of retaining multiple versions of pages, we have one version of each page; particular locations may hold multiple versions of their contents (see Figure 6). This allows the pages to be accessed through a virtual memory page table (providing the illusion of a flat single-timestep address space), as described in section 3.4 Many memory operations must check for a forwarding pointer, however, and sometimes execute a history stack operation.

6. PERFORMANCE PROJECTIONS

Until very recently, predicting demonic memory's performance was impossible; the necessary data were not available. Most other data on Lisp programs are gathered using synthetic benchmarks (e.g., [Gabr85]). While these are useful for testing particular aspects of Lisp systems, they do not give a realistic picture of the behaviors relevant to our system. Some other data have been published (e.g., [ClGr77]), but they do not include the necessary information for our purposes.

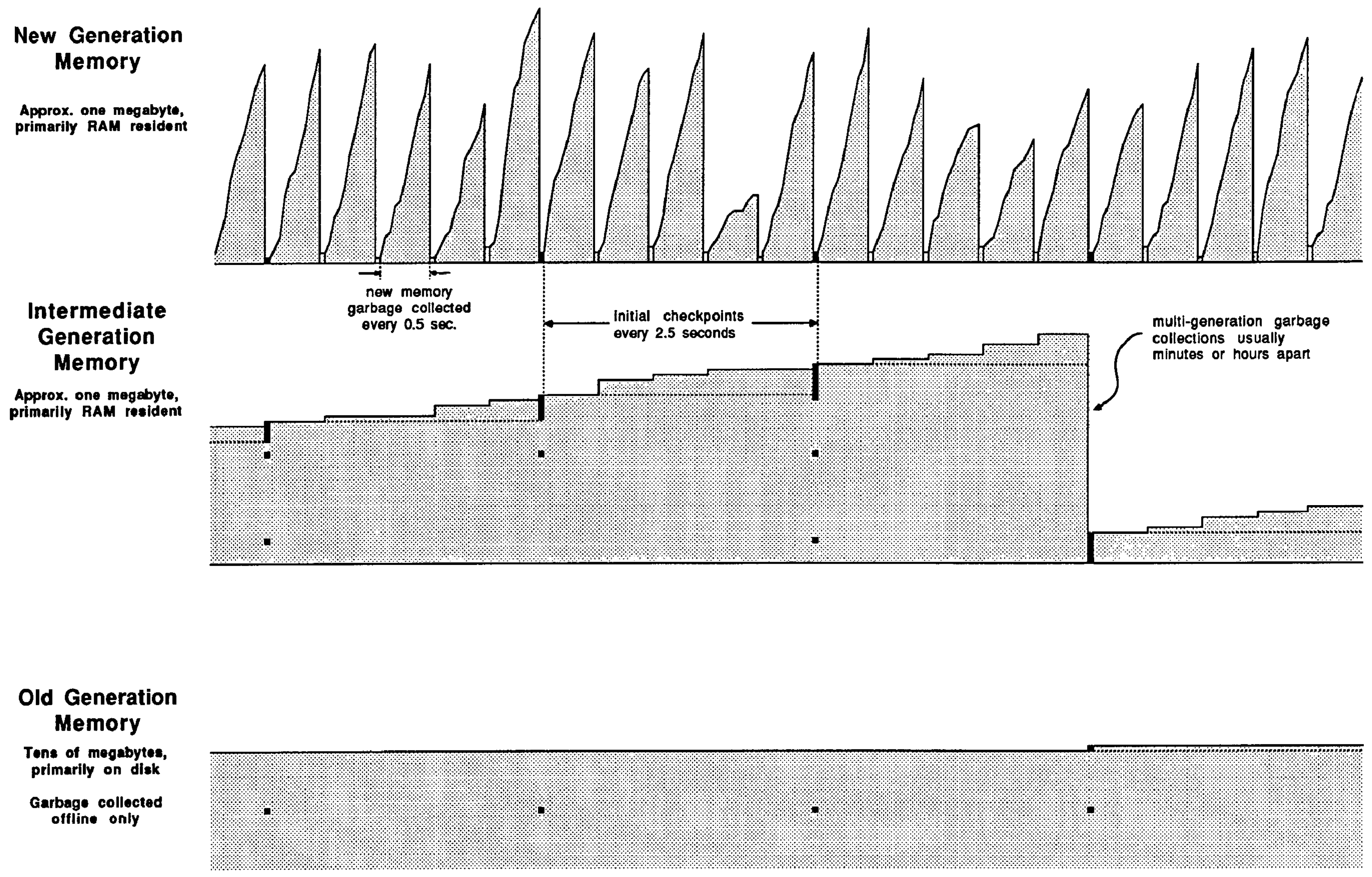

New Generation Memory
Approx. one megabyte, primarily RAM resident
new memory garbage collected every 0.5 sec.
Intermediate Generation Memory
Approx. one megabyte, primarily RAM resident
Initial checkpoints every 2.5 seconds
multi-generation garbage collections usually minutes or hours apart
Old Generation Memory
Tens of megabytes, primarily on disk
Garbage collected offline only

FIGURE 5. Pattern of memory usage and its relationship to checkpointing.

Checkpoint snapshot space costs can be broken down into several components, as shown here; new data that survive garbage collection, aging data that have recently been copied into older generations (i.e., above horizontal dotted lines), and other pages that have been modified since the previous checkpoint (all are shown in black).

Very Old generation memory (not shown) may be very large, and is seldom if ever garbage collected.

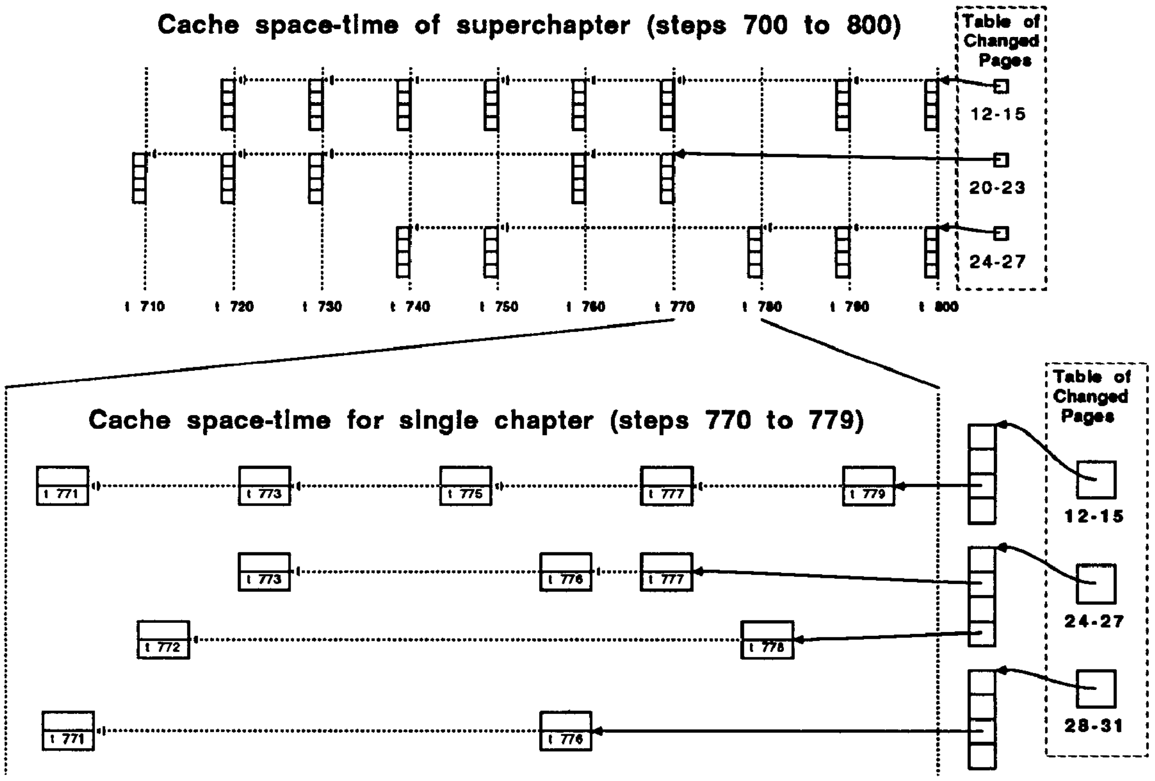


FIGURE 6a. A cache spacetime at single-step resolution.

When it is necessary to access the process history at single-step resolution, a special kind of cache spacetime is generated.

Only a single page-version is generated for each changed page. If a location only takes on one value during the chapter, that value is stored directly in the appropriate location in the appropriate page. A second store into a location causes the original value to be replaced by a special forwarding pointer to a history stack for the location. This history stack will contain records of all changes to the location within the chapter. Thus each location contains either its only value during the chapter, or a pointer to a stack that can be searched for its value at any step. (See Figure 6b, below.)

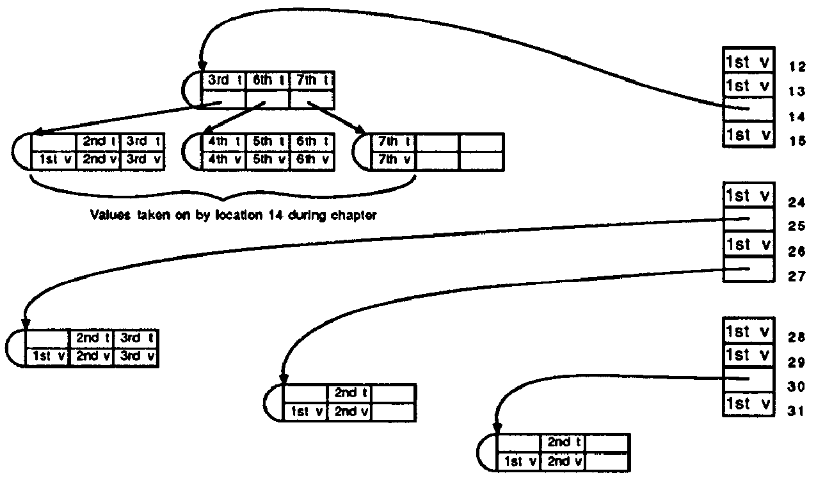


FIGURE 6b. Detail of hi-res cache.

As before, the history stacks are actually the tree structures of Figure 1.

Fortunately, Shaw has recently published data that allow us to make ballpark estimates [Shaw88]. Shaw's statistics were gathered for runs of four large compiled Common Lisp programs; their execution times range from several seconds to several minutes on a 4 MIPS processor. While this is a small number of data points, the programs were carefully chosen to be varied and realistic. At any rate, they are the best data available.

For our projections, we assume a processor that runs Lisp four times as fast as Shaw's. This needn't actually be a 16-MIPS machine, if the hardware is more amenable to fast Lisp execution. We see this as reasonable performance for a high-end workstation in the next few years.

The target configuration is one that is intended to provide high-performance demonic memory for a monolingual software development environment on a high-end workstation. Our goal is to allow access to any previous state of the system within 6 or 7 minutes (even states that may have occurred months or years previously). More recently-accessed states should be accessible much more quickly. We assume that two half-hour stretches of history are always available at higher resolution, so that any state within them can be accessed within a minute or two. Four five-minute stretches can be kept at still higher resolution, so that any state within them can be reconstructed within about fifteen seconds. Finally, a few half-second intervals can be retained at single-step resolution, allowing instant access to any step within them (approximately 8 million machine instructions each.)

To support such performance, we use a demonic memory with five levels of resolution, ranging from single-step to super^{4}chapters. Garbage collection of new memory and chapter bounding occur at half-second intervals. History is not initially retained at this resolution, however; initial checkpoints occur at 2.5 second intervals, or superchapter resolution. The initial snapshots are kept in RAM only; super^{2}chapter snapshots are saved to magnetic disk every 10 seconds. Most of these snapshots are discarded, to reduce resolution to about a minute, or super^{3}chapters. At five-minute intervals, super^{4}chapter snapshots are saved to WORM disk for permanent storage.

The copying and space costs of such a system fall into three major categories: changed stack and symbol table values, new data objects that have survived garbage collection, and data in older generations that have been changed. [Shaw88] shows how to compute very similar costs in predicting the performance of his garbage collector for Lisp. (Following Shaw, we assume that symbol table organization is favorable to our scheme, being organized as parallel vectors rather than an array of records.) Due to space constraints, we will only summarize our results here; detailed performance projections are available from the authors on request [Wils88a].

Approximately four extra megabytes of RAM, eleven megabytes of magnetic disk, and a write-once optical disk drive will support the specified performance level; blank WORM disks will be consumed about every six months at 40 hours per week of compute-bound computation, or several *years* for the average user. The time costs should be under 10 percent of normal running time, including trap-handling time, etc. Another reasonable configuration would use fifteen megabytes of RAM and eliminate the use of magnetic disk entirely. This would eliminate most disk access time and greatly reduce wear and tear on moving parts.

Some caveats are in order here. These projections (like Shaw's) do not include a cost that we consider potentially important — costs due to changed pages of data that have been creating during the execution of a program, but which have survived to be copied out of the newer generation(s). If locality is poor in such data, the cost could be significant. It may be necessary to use locality-increasing techniques to cluster written-to objects onto a small number of pages. These techniques will themselves incur additional costs.

Our garbage collector can easily support such a policy due to peculiarities in its mechanism for recording intergenerational pointers [WiMo89a]. This scheme will also allow some easy optimizations to reduce storage costs. With these techniques, the basic costs should not go beyond the costs stated above, unless write locality is extremely bad.

This garbage collector design incurs a continual runtime speed overhead of a few percent, however, when compared to some other advanced designs (e.g., [Moon84], [Shaw88]). This could raise the total time overhead for demonic memory to around fourteen percent. On the other hand, our garbage collector's improved locality may pay for itself in reduced paging during garbage collection, irrespective of demonic memory. If it pays for its overhead simply by reducing backing store operations, the time cost of demonic memory (over and above the cost of garbage collection) should still be under 10 percent.

It should be borne in mind that these speed costs are relative to the speed of Shaw's compiled Common Lisp. (His system performed only basic optimizations such as tail-recursion elimination and inlining of the most frequent operations.) For faster systems, the overhead will be proportionally higher, while it may be considerably lower for slower systems such as existing Smalltalks.

7. RELATED WORK AND APPLICATIONS

Our system shares important features with other systems, both at the implementation level and at the language level.

7.1 Language constructs for recovery and tentative execution

Demonic memory would be an appropriate mechanism for the **probe**, **try**, **guard** and **undo** recovery constructs advocated by Heering and Klint [HeKl85] for integrating command, programming, and debugging languages in a monolingual programming environment. We plan to use demonic memory to implement undo, allowing a process to revert to any previous state. (We may not use it for probe, try, or guard, however, because we have devised another mechanism, *alternate universes*, that may be more suitable [Wils88a] in general. Alternate universes are virtual copies of a program's entire execution environment, allowing multiple threads of computation to proceed without being affected by each others' side-effects. This allows a fine-grained multiway generalization of side effect recovery constructs, or *side effect isolation.*)

Demonic memory can also be used to implement tentative execution constructs for backtracking or lookahead, such as those those available in Interlisp [Teit81] and T-Pascal [StCo87]. It is considerably more efficient in the large, however, and more comprehensive than the facilities in other languages such as Icon [Gris83]. Demonic memory could also support constructs for software fault tolerance [Rand75].

Note that while most language-level recovery constructs are variants of "undo," ours is more precisely a "revert" command. Conceptually, past states are *never* thrown away, even if we "undo" to an earlier state. The current state may be transformed into a past state by bringing the old versions of data up to the present, without discarding the intervening information. (Our recovery is much like Leeman's phi operator [Leem86] or "undo forward" [Leem85]. Where Leeman's recovery applies to a specified set of variables, ours applies by default to the whole program state. Ours has a larger scope in time as well, allowing reversions to long-past states rather than for some limited number of steps.) A single control construct providing this facility can be used to implement all of the recovery constructs we have mentioned, as well as many others.

This construct, **call-with-captured-state**, is a very general escape mechanism, much like Scheme's call-with-current-continuation. Call-with-current-continuation (or "call/cc") captures the current control context (continuation) when it is executed, usually by saving the stack of activation records. This control context is packaged up into an escape procedure. Later execution of this escape procedure will restore the original control context (e.g., activation stack); this effectively causes a jump back to the control point at which call-with-current-continuation was executed. Thus call/cc can be used to implement a variety of control constructs, including backtracking [FrHK84].

In contrast, call-with-captured-state (or "call/cs") captures *all* state at the time of its execution, not just the control information. If the escape procedure is executed, the captured state is restored. (This is achieved by restoring all pages to their versions at that point. It generally entails reconstructing from the previous checkpoint, replaying the relevant segment of the exception list.)

Both call/cc and call/cs allow the escape procedure to be executed any number of times. It is also a first-class object that can be passed around like any other, with indefinite extent. Once a computation has been escaped from, it is thus possible to escape back into it and resume the computation. In the case of call/cc this allows the implementation of control constructs such as coroutines entirely within the Scheme language [HaFW84]. Our call/cs will similarly enable nonlinear version maintenance, such as Vitter's "undo, skip and redo" [Vitt84], which generates a tree topology.

It should be noted that call/cs has the potential to cause a performance problem if usage patterns are malign. Regeneration a previous state may itself cause re-execution of a previous escape, in turn causing regeneration of *another* previous state. Such "chained" recoveries could in the worst case be equivalent to re-executing an entire process. This is not an issue for most applications, which require only simple recovery; e.g., for reversible debugging of conventional programs. We also expect a caching scheme to avoid most chained rollbacks at moderate costs when call/cs is used freely as a language construct. For some applications however, it may be preferable to actually truncate a space time to restore a previous state, rather than retaining the intervening history as in Leeman's "undo backward".

In addition to call/cs, we will eventually incorporate a construct that allows reverting to *any* previous state (rather than particular "control points" created by call/cs). This relatively unstructured, low-level capability would allow the implementation of features like stepping debuggers within the language. It is unclear at this time what the details of this construct should be, to allow the maximum clarity, flexibility and efficiency; we expect to do considerable experimentation before settling on a particular scheme.

7.2 Checkpoint and replay systems

Demonic memory uses full, not partial checkpointing (in the sense of [ArCS84]), since the entire state of the system is saved at each checkpoint. Most of the efficiency of partial checkpointing is achieved instead through copy-on-write virtual copy techniques [cf. MiBK86, Rash87, StYB88]. Looked at another way, however, every non-checkpoint past state may be regarded as a partial checkpoint — the exception list and history-generating program form the "log" of changes required to transform a fully checkpointed state (snapshot) into a partially checkpointed one. Note that demonic memory keeps the intervening history fully checkpointed at some resolution. When a chapter has been fully fleshed-out at single-step resolution, for instance, *every* step within it is fully checkpointed.

The particular checkpointing and reconstruction techniques used in demonic memory are most similar to those used in some recovery systems for distributed processing (e.g., [BoBG83, PoPr83, StYe85]). In these systems, states are checkpointed and interprocess communication is recorded for later replay, much like our virtual snapshots and exception lists. These systems retain only a small amount of history, while demonic memory's hierarchical checkpointing retains long histories at some resolution. Demonic memory allows random access to states, not just (forward) replay. It thus also supports reverse execution facilities [Balz69, TeRe81, Mohe88]. It is in several ways similar to Feldman's IGOR debugging system [FeBr88], though coordination with garbage collection should yield much better performance due to increased effective locality.

Checkpointing and message replay mechanisms have been used for recovery [StYe85, StYB88], and to allow concurrent processes to be replayed independently for debugging (e.g., [CuWi82, PaLi88]). A distributed version of demonic memory could provide such a facility, while allowing jumping or stepping forward or backward to arbitrary steps of the process, and efficiently supporting dynamic data structures in a garbage-collected heap.

Demonic memory's hierarchical checkpointing and reconstruction bears an interesting similarity to Miller and Choi's *incremental tracing* [MiCh88]; both allow regeneration of program history at multiple levels of resolution. Their system uses sophisticated static analysis to determine which variables may be changed during the execution of each block of code, however, while ours relies on dynamic detection of changes to parts of memory. Miller and Choi's system exploits its analysis of code in performing *flowback analysis* [Balz69] and in guaranteeing deterministic re-execution of parallel programs operating on shared memory. Their system does not currently support heap data, however. (It may be difficult to extend such techniques to pointer data while preserving efficiency, especially for dynamically-typed languages and those with first-class or higher-order procedures.)

7.3 Optimistic systems

Optimistic techniques involve the "premature" computation of results that may or may not turn out to be useful (e.g., [KuRo81], [Jeff85]). Rather than blocking while waiting for the result of an interprocess interaction, a process guesses what the outcome will be and proceeds on that assumption. If it guesses wrong, it may have to "roll back" to the error and re-execute when the true outcome is known. If the guess is right, it has increased the effective parallelism of the computation. Thus, processors that would otherwise be idle can perform potentially useful computation.

Optimistic techniques work well for asynchronous processes where interactions usually fail to occur (as in distributed simulation), and other relatively predictable situations. Like demonic memory, optimistic systems require the ability to return to a previous state, and employ checkpointing and "replay" mechanisms. Demonic memory's efficiency in handling heap data should make it attractive for such applications; current optimistic systems generally restrict the use of dynamically allocated data or disallow it completely [e.g., StYe85, Jeff87].

Optimistic systems for executing Lisp programs have previously been proposed by Knight [Knig86] and Katz [Katz86]. These designs are intended to execute overtly serial programs optimistically in parallel. We envision a different sort of optimistic system, in which the parallelism is explicit, using demonic memory and alternate universes to implement parallel control structures. This would favor an optimistic computational style, much in the spirit of Halstead's *speculative* computation [Hals86]. Unlike speculative computation, however, optimistic computation would still be completely deterministic, with all of the corresponding advantages that determinism yields for debugging and verification.

Smith and Maguire's ***multiple worlds*** facility supports optimistic and speculative computation using pagewise copy-on write virtual copying for ***side-effect isolation*** [SmMa89]. Our own ***alternate universes*** design performs a similar function, though it is finer grained and coordinated with garbage collection. These systems support a tree topology of states; recovery is fast and efficient, but limited to branch points rather than arbitrary process steps.

7.4 Production systems and inference systems

Production systems and inference systems are frequently used in artificial intelligence and cognitive modeling, but they tend to be difficult to use and debug. Many systems provide only rudimentary trace functions giving dependencies among the rule firings (or resolutions) and their effects. We would often like to examine some prior state of the system's working memory, not only to find out what caused a rule to fire, but also to understand why other rules did not. (Roughly, we may want to know what information was missing such that some inference was *not* made.) This "negative" information is lost in current systems, but could be made available through demonic memory.

8. CURRENT STATUS

We are currently building a prototype of demonic memory, with a subset of the features we have described. This prototype does not actually use any special virtual memory techniques, but does allow simulating them for gathering statistics. As in IGOR [FeBr88], checkpointing is accomplished by flushing dirty pages at checkpoints, rather than by copy-on-write. (This requires some redundant memory so that information is not lost when a page is written to, but allows simulating various page sizes.) The prototype also uses copying of data rather than page remapping, which slows some operations but is acceptable for a prototype dealing with small-to-medium-sized address spaces.

We are currently using the Scheme-48 language processor, a bytecoded implementation of Scheme [StSu75,ReCl86] developed by Jonathan Rees (MIT), Richard Kelsey (Yale), and Bob Brown (MIT). We have implemented and instrumented our own generational garbage collector, with several innovations for high efficiency [Wils88b, Wils89a, WiMo89a, WiMo89b].

We intend to port this garbage collector to other language processors to gather statistics on more and larger programs than will run in our prototype. We will evaluate demonic memory's efficiency using a combination of statistics from the running prototype and relevant data from other systems. By scaling appropriately for our system's deficiencies, we should get a fairly accurate view of the true costs of demonic memory for high-performance systems.

9. FUTURE WORK

After gaining experience with our prototype, we must address several important issues that arise in a real high-performance systems. These include efficient support for the forwarding pointers used during single-step regeneration, and the maintentance of the notion of a "step" when no interpretive instruction counter is available. We also wish to address issues that arise in parallel systems.

Single-step regeneration (to record timestamps) is difficult with compiled code, because the code must check for forwarding pointers when fetching and storing values, such checks should not slow the system down in normal running, however. If parallel hardware support and/or microcoding is available (as on Lisp machines), the situation is greatly simplified. If not, several other strategies are possible.

One solution is to replace the compiled fetch and store instructions with subroutine jumps, much like breakpoint insertion in a debugger; subroutines can then do the checking and tree-accessing operations. This entails finding all of the relevant code and modifying it before it is re-executed, however. This can be accomplished fairly easily in some systems (e.g., those with a small cache of throwaway-compiled code). In other systems it is more complex; it may involve access-protecting pages containing machine code so that traps can insert subroutine jumps into each page that is actually accessed by the running program.

The stepping problem is similarly simplified if hardware support is available; e.g., a special counter registers for debugging purposes, as has been advocated and/or implemented by others. Failing that, several software approaches are available. The crudest is to make the same distinction as is conventionally made in Lisp systems — interpreted code is steppable, but compiled code executes atomically. Alternatively, we might actually compile in operations on a counter, most of which could be optimized out again. Or we could use the heap allocation pointer as an kind of crude counter between scavenges, making it do double duty. (This would allow halting the program at an arbitrary "high-water mark" allocation operation during regeneration, by setting the allocation limit pointer at that height.)

For the present, we assume that truly asynchronous hardware interrupts are handled by low-level code, insulating the program-level notion of "states" from changes that occur "between" steps. Higher-level interrupts are assumed to be handled only at step boundaries. This seems appropriate for a high-level programming environment, but we are exploring alternative strategies for a comprehensive monolingual system.

Eventually, we hope to parallelize demonic memory, in two senses. First, we would like to make a demonic memory that implements a single space-time across several processors. Second, we would like to generalize demonic memory's linear sequence of states to an efficient tree topology like that of multiple worlds or alternate universes. This would result in a branching *space-time manifold*, allowing access to any past state along any path.

10. CONCLUSIONS

Demonic memory supports a comprehensive recovery facility allowing fast access to past states of a computational process. It supports a space-time abstraction in which memory is viewed as a two-dimensional virtual array, or space-time. A *reconstructive memory* technique allows long histories to be stored sparsely, regenerating detailed information on demand. A copy-on-write *virtual copying* scheme is used for checkpointing, with sophisticated indexing and caching allowing fast access to stored states without undue space costs. *Hierarchical checkpointing* allows fast reconstruction at several levels of time resolution. Most importantly, coordination with garbage collection allows checkpointing of heap data at a moderate cost.

ACKNOWLEDGMENTS

We would like to thank the many people who have influenced this work. In particular, our understanding of garbage collection issues was advanced by informative discussions with Henry Lieberman, David Ungar, Patrick Caudill, David Moon, and Patrick Sobalvarro; special thanks to Bob Shaw, whose data and analyses have been invaluable. We also thank the program committee for their careful reading and detailed suggestions.

REFERENCES

ArCS84 Archer, J.E., Conway, R., and Schneider, F.B. User recovery and reversal in interactive systems. *ACM Trans. Prog. Lang. Syst, 6*, 1 (January 1984), pp, 1-19.

Balz69 Balzer, R.M. EXDAMS — Extendable debugging and monitoring system. In *AFIPS Proceedings Spring Joint Computer Conference* (Boston, Mass., May 14-16), 34, AFIPS Press, Arlington, Va. 1969, pp. 567-580.

BoBG83 Borg, A., Baumbach, J., and Glazer, S. A message system supporting fault tolerance. *Proc. Ninth ACM Symp. on Operating Systems Principles*, October 10-13, 1983, pp.100-109.

ClGr77 Clark, D. and Green, C. An empirical study of list structure in Lisp. *Comm. ACM 20*, 2 (February 1977).

CuWi82 Curtis, R. and Wittie, L. BugNet: a debugging system for parallel environments. *Proc. 3rd Int'l. Conf. on Distributed Computing Systems* (October 1982).

FeBr88 Feldman, S. and Brown, C. IGOR: A System for Program Debugging via Reversible Execution. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 112-123.

FrHK84 Friedman, D.P., Haynes, C.T., and Kohlbecker, E. Programming with continuations. In P. Pepper, ed., *Program transformation and programming environments*, pp. 263-274. Springer-Verlag, 1984.

Gabr85 Gabriel, R. *Performance and evaluation of Lisp systems*. MIT Press, Cambridge, Mass., 1985.

Gris83 Griswold, R.E. and Griswold, M.T. *The Icon programming language*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

Hals86 Halstead, R. Parallel symbolic computing. *Computer 19*,8 (August 1986), pp.35-43.

HaFW84 Haynes, C.T., Friedman, D.P., and Wand, M. Continuations and coroutines. *Proc. 1984 ACM Symposium on Lisp and Functional Programming*. August 6-8, 1984, pp. 293-298.

HeKl85 Heering, J., and Klint, P. Towards monolingual programming environments. *ACM Trans. Prog. Lang. Syst., 7*, 2 (April 1985), pp. 183-213.

Jeff85 Jefferson, D.R. Virtual Time. *ACM Trans. Prog. Lang. Syst., 7*, 3 (July 1985), pp. 404-425.

Jeff87 Jefferson, D.R., *et al.* Distributed simulation and the Time Warp Operating System. *Proc. 11th ACM Symposium on Operating Systems Principles* (November 1987).

Katz86 Katz, M.J. ParaTran: A transparent, transaction based runtime mechanism for parallel execution of Scheme. Masters thesis, Massachusetts Institute of Technology, June 1986.

Knig86 Knight, T. An architecture for nearly functional programming. *Proc. 1986 Conf. on Lisp and Functional Programming*.

KuRo81 Kung, H.T., and Robinson, J.T. On optimistic methods for concurrency control. *ACM Trans. on Database Systems 6*, 2 (June 1981).

Leem85 Leeman, G.B. Building undo/redo operations into the C langauge. *Proc. 15th Annual Int'l. Symp. on Fault-Tolerant Computing*. IEEE Computer Society Press, 1985, pp. 410-415.

Leem86 Leeman, G.B. A Formal Approach to Undo Operations in Programming Languages. *ACM Trans. on Programming Languages and Systems 8*, No. 1, January 1986, pp. 50-87

LiHe83 Lieberman, H., and Hewitt, C. A real-time garbage collector based on the lifetimes of objects. *Comm. ACM 26*, 6 (June 1983), pp. 419-429.

MiBK86 Mittal, S., Bobrow, D. and Kahn,K. Virtual Copies: at the boundary between classes and instances. *Proc. OOPSLA '86*. (September 1986), pp. 159-166.

MiCh88 Miller, B.P. and Choi, J.D. A Mechanism for Efficient Debugging of Parallel Programs. *Proc. ACN SIGPLAN Conf. on Prog. Lang. Design and Implementation* (June 1988), pp.135-144.

Mohe88 Moher, T.G. PROVIDE: A process visualization and debugging environment. *IEEE Trans. Software Engineering 14*, 6 (June 1988), pp. 849-857.

Moon8 Moon, D. Garbage collection in a large Lisp system. *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984, pp. 235-246.

PaLi88 Pan, Douglas Z., and Linton, Mark A. Supporting reverse execution of parallel programs. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 112-123.

PoPr83 Powell, M.L., and Presotto, D.L. Publishing: a reliable broadcast communication mechanism. *Proc. Ninth ACM Symp. on Operating Systems Principles*, October 10-13, 1983, pp.100-109.

Rand75 Randell, B. System Structure for software fault tolerance, *IEEE Transactions on Software Engineering* vol. SE-1, no. 2, pp. 220-232 (June 1975).

Rash87 Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*. October 4-8, 1987. pp. 31-39.

ReCl86 Rees, J., and Clinger, W. The revised^3 report on the algorithmic language Scheme. MIT AI Memo 848a, September, 1986..

Shaw88 Shaw, R. Empirical analysis of a Lisp system. Stanford University Ph.D. thesis, CSL-TR-88-351, February 1988.

SmMa89 Smith, J.M., and Gerald Q. Maguire, Jr. Transparent concurrent exectution of mutually exclusive alternatives. Ninth Int'l. Cof. on Distributed Computer systems, Newport Beach, California, June 1989. Also available as Columbia University tech. report CUCS-387-88.

StCo87 Strothotte, T.W., and Cormack, G.V. Structured program lookahead. *Comput. Lang. 12*, 2 (1987), pp. 95-108.

StSu75 Steele, G. L., and Sussman, G. J. Scheme: an interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory, 1975.

StYe85 Strom, R.E., and Yemini, S. Optimistic recovery in distributed systems. *ACM Trans. Comp. Syst. 3*, 3 (August 1985).

StYB88 Strom, R.E, Yemini, S, and Bacon, D. A recoverable object store. Technical report, IBM T.J. Watson Research Center. Submitted for publication.

TeRe81 Teitelbaum, T., and Reps, T. The Cornell Program Synthesizer: a syntax directed programming environment. *Comm. ACM 4*, 9 (September 1981), pp. 563-573.

Teit81 Teitelman, W. Automated programmering: The Programmer's Assistant. In Barstow, Shrobe, and Sandewall (eds.) *Interactive programming environments*. New York: McGraw-Hill, 1981, pp. 232-239.

Unga84 Ungar, D. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. *Proc ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984. pp. 157-165.

Unga86 Ungar, D. *Design and evaluation of a high-performance Smalltalk system.* MIT Press, Cambridge Mass., 1986.

Vitt84 Vitter, J.S. US&R: a new framework for redoing. *Proc. ACM SIGSOFT/SIGPLAN Software engineering symposium on practical software development environments (Pittsburgh, PA, April 23-25).* ACM Publications, New York, 1984, pp. 168-176.

Wils88a Wilson, P. R. Two comprehensive virtual copy mechanisms. Masters' thesis, University of Illinois at Chicago EECS department, 1988.

Wils88b Wilson, P.R. Opportunistic garbage collection. Forthcoming in *SIGPLAN Notices*.

Wils89a Wilson, P.R. A simple bucket-brigade advancement mechanism for generation-based garbage collection. Forthcoming in *SIGPLAN Notices*.

WlMo89a Wilson, P.R. and Moher, T.G. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. Forthcoming in *SIGPLAN Notices*.

WlMo89b Wilson, P.R. and Moher, T.G. Design of an Efficient Generational Garbage Collector. Submitted to *OOPSLA '89*.