A SOFTWARE CACHE MANAGEMENT SYSTEM

by

Jeffrey N Eisen

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE

in partial fulfillment of the requirements

FOR THE DEGREES OF

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1985

Signature of Author_____
Department of Electrical Engineering and Computer Science, May 9, 1985

Certified by_____
Dr. Richard Eliot Zippel, Thesis Supervisor

Certified by_____
Dr. Craig Warren Thompson, Company Supervisor

Accepted by_____
Dr. Arthur Clarke Smith, Chairman,
Departmental Committee on Graduate Students

A SOFTWARE CACHE MANAGEMENT SYSTEM

by

JEFFREY N EISEN

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 1985 in partial fulfillment of the requirements for the degrees of
Master of Science and Bachelor of Science in Computer Science

ABSTRACT

Caching of data is a commonly used mechanism for increasing the efficiency of computer programs. It is often much more efficient to store objects when they are first computed and reuse the cached objects when they are needed in the future, than it is to recompute the objects every time they are needed. Existing methods for caching data are successful in increasing program performance in special cases, but are not broadly applicable.

The focus of this thesis is SCMS, a Software Cache Management System. SCMS provides a mechanism designed to improve the performance advantage of using software caches by exploiting general properties of cached information. The most important such property to be considered is reconstructibility. SCMS provides a simple interface that enables the application programmer to specify "cost functions" and "action functions" that enable SCMS to make knowledgable decisions regarding the validity and usefulness of caches. This knowledge is used to accomplish two tasks: ensure the validity of cached information seen by the application program and improve memory utilization by purging cached objects that are not cost-effective.

Thesis Supervisor:	Richard E. Zippel
Title:	Assistant Professor, Computer Science and Engineering
Company Supervisor:	Craig W. Thompson
Title:	Senior Member of Technical Staff, Texas Instruments

Acknowledgments

I would like to thank Craig Thompson and Richard Zippel for invaluable guidance, Texas Instruments for providing facilities and patience, Jack Florey for keeping me sane and driving me insane, and my family for almost everything else. Special thanks to Donald Knuth for inventing TEX, a miraculous text formatter, without whose help this might never have been completed.

Contents

List of Figures

Chapter 1

Introduction

Caching of data is a commonly used mechanism for increasing the efficiency of computer programs. It is often much more efficient to store objects when they are first computed and reuse the cached objects when they are needed in the future, than it is to recompute the objects every time they are needed. Although existing methods for caching data are successful in increasing program performance, the level of success is far from optimal. The focus of this thesis is SCMS, a Software Cache Management System. SCMS provides a mechanism designed to improve the performance advantage of using cached information by exploiting general properties of cached information.

In this thesis, a cache is defined to be any sort of information (for example, data or code) that is saved because it may be reused later, even though it can also be recomputed if necessary. That is, cached information is never needed; it can always be recreated because it is redundant.

Though almost all software systems use some form of caching, caching is almost always done in an ad hoc fashion. There are two primary disadvantages of this ad hoc approach to caching when compared to a standard caching interface that is used for all types of applications. First, whenever a new application program re-

quires caching for any reason, almost all of the code related to caching needs to be written. This introduces unnecessary work for the application programmer, which may result in information not being cached because it is not worth expending the effort necessary to write the code. The extra code written to do the caching may obscure the application's algorithm and may introduce "bugs" into the application program. Second, these ad hoc schemes for caching information often do not take advantage of general properties of cached data; specifically, they do not take advantage of the fact that cached data can be recreated if necessary. They provide no capability to independently specify how and when cached objects are to be created, modified, destroyed, or otherwise affected.

There are two main tasks of a software cache management system: ensuring cache validity and managing storage. Although the increase in performance gained from using cached data is clearly desirable, one must be careful when using cached information. If some aspect of the environment has changed since the time that the cache was originally created or last modified, then cached results may no longer be valid. For example, in a VLSI schematic entry system, where schematics are repeatedly displayed on a graphics terminal, there might well be a caching mechanism to eliminate the need to recompute the bitmap each time a transistor is drawn. However, if the function that draws transistors is modified in order to change the graphical representation of a transistor, then the cached bitmap is no longer valid.

A problem shared by most of the commonly used caching mechanisms is that there is no way of knowing that there is invalid information present until an attempt is made to use the data. This problem results in the address space being unnecessarily filled with invalid data, leaving less space available for valid, more important data. It would be advantageous if, once a cached object becomes invalid, the memory that it occupies becomes freed for garbage collection purposes

automatically, not just the next time an attempt is made to use the cached data. Although this observation does not always hold (sometimes, it is useful to preserve invalid information for purposes such as debugging), purging invalid information in order to free memory is an important concern in a number of situations.

It may also be desirable that valid cached objects may be released to be garbage collected. Such objects are said to be superfluous. Superfluous, in this context, means that it is determined that the cost (in memory, general overhead, etc.) of keeping the object cached is greater than the expected performance gain of retaining the cached object. Cost functions, used to determine superfluousness, associated with recreation, destruction, and other operations, can be used to tune performance.

In this thesis, a software cache management system, SCMS, is presented which is responsible for the two tasks described above: ensuring the validity of cached objects and automatically allocating and deallocating memory for these objects. SCMS provides a simple interface to application programs that enables the programmer to better utilize cached information. SCMS is implemented on Lisp Machines (Explorers, Symbolics, and others); however, the approach used in designing SCMS can be applied to a wide range of computer architectures.

In Chapter 2, the goals of a software cache management system are described. The advantages and disadvantages of related mechanisms are presented with respect to these goals. SCMS is described and generalized example SCMS interfaces are presented. Chapter 3 presents applications in which SCMS has been installed and describes additional applications which would benefit from using SCMS. Finally, Chapter 4 presents conclusions of this research and directions for future research in this area.

Chapter 2

SCMS

This chapter describes SCMS, a system for managing cached information on Lisp Machines. Section 2.1 presents guidelines that were used in designing SCMS. Section 2.2 presents five related systems with emphasis on how these systems accomplish the two tasks of a software cache management system described in Chapter 1 with respect to the guidelines presented in Section 2.1. Section 2.3 describes the implementation of SCMS. Finally, Section 2.4 presents generalized SCMS interfaces for the two primary types of domains appropriate for SCMS operation.

2.1 Guidelines for a Software Cache Management System

A cache management system is defined to be "a tool designed to better utilize cached information." This definition is general and, in Section 2.2, several mechanisms are presented that come under the scope of this definition. In designing SCMS, three more specific properties of a software cache management system were identified: transparency, safety, and non-interference.

2.1.1 Transparency

Ideally, a system that manages cached information should be completely transparent to the application program. There are two areas where transparency is important.

The first area is in defining the application. Complete "definition transparency" means that the cache management system would be able to:

1. determine which objects need to be cached,
2. determine the validity of cached objects,
3. determine relative importance of cached objects, and
4. take appropriate actions based on this determination,

all without any application specific assistance. In other words, in a completely definition transparent system, *there would be no need to define a caching interface for an application*. The cache management system would be powerful enough to "understand" all application specific information without any application specific assistance.

All four of these steps are difficult problems making complete definition transparency a difficult goal to achieve. Step 1 means that the cache management system, without any application specific assistance, would be able to decide which objects are both likely to be used again and costly enough to create, to be good candidates for caching. Step 2 implies that a general cache manager would be able to determine cache validity. Step 3 implies that application independent cost functions could be written to govern relative utility of caches. Step 4 means that the cache management system would be able to delete invalid or superfluous objects from the address space, but still be able to reconstruct the objects if they were requested by the application program.

The second area where transparency is important is in the use of the cached data. Full "use transparency" means that no special care needs to be taken when cached information is used within an application program and that there is no extra syntax associated with cached data. For example, if the variable *cached-variable* holds cached information, then statements such as

```
(setq other-variable cached-variable)
```

and

```
(setq another-variable (car cached-variable))
```

could be used freely. These examples bring up both performance and semantic considerations involving use transparency, such as:

1. Is *other-variable* bound to *cached-variable*'s current value, or is *cached-variable* validated first? That is, is *cached-variable* checked for validity (and modified if it is invalid) before *other-variable*'s value is set?
2. Is *another-variable* bound to the **car** of *cached-variable*, or is *cached-variable* validated before the **car** is taken?
3. Does *other-variable* become a cached variable; that is, in the future, will the value of *other-variable* change as the value of *cached-variable* changes (due to explicit **setq**s or due to the implicit actions of the cache management system)?
4. If *cached-variable* is always validated first, must all calls to **setq**, **car**, **eq**, and other commonly used functions, pay this performance penalty?

Full use transparency can be achieved by deciding upon a semantics for using cached objects such as "validate cached objects before each use; do not allow pointers into the middle of cached objects." However, achieving full use transparency may involve paying an unacceptably high performance penalty.

2.1.2 Safety

SCMS is conceptually similar to a virtual memory paging scheme. However, in a paging scheme, objects are only swapped out of the real memory when they become old; in SCMS, they are completely removed from the virtual address space. The complete removal of cached objects from the virtual address space has the potential of corrupting the address space with "dangling" or illegal pointers.[1]

A basic guideline used in the design of SCMS is that, regardless of whether or not the application program uses SCMS "properly," there should be no way to accidentally corrupt the Lisp Machine environment. If the user abides by SCMS protocols, then correct behavior should be expected. If, on the other hand, the user does not abide by SCMS protocols, then no guarantees of correct behavior will be made. However, none of the incorrect behavior will be "dangerous"; incorrect values may be returned, but no illegal pointers will ever be created. To avoid this danger, SCMS must ensure that, in the creation and destruction of pointers to a cached object, no dangling pointers remain after the cached object is purged.

2.1.3 Non-Interference

A final consideration in the design of SCMS was to try to ensure that SCMS does not interfere with system operations like the garbage collection of objects that would otherwise be considered garbage. Ideally, garbage collection of an object would not be prevented if the only pointers to the object are those that SCMS uses to keep track of the object. This is difficult because of the following contradictions. In order for SCMS to know about a cache, there must be a pointer to the cache. It does not suffice just to retain the virtual memory address of the cache because the cache's

[1] According to the Lisp Machine Manual [10], illegal pointers are "pointers, that are, for one reason or another, according to storage conventions, not allowed to exist."

address may change due to actions of the garbage collector. On the other hand, in order for an object to be garbage collected, there must not be any pointers to the object. SCMS is not successful in achieving this goal; however, in Chapter 4, a possible approach to this problem is briefly described.

2.2 Related Systems

Using the above definition of a cache management system — "a tool designed to better utilize cached information" — several commonly used mechanisms can be considered to be special cases of cache management systems including timestamping, garbage collection, resources, virtual memory paging, and explicit storage control. Each mechanism addresses part of the cache management problem, but fails to address the entire problem.

2.2.1 Timestamping

Timestamping is a method that can be used to ensure the validity of cached information. Timestamping involves marking the cached data with a timestamp, which is some notion of the time that it was created, last used, or last determined to be valid. Then, when the cached information is later needed, a predicate is applied to the timestamp to determine, in some manner, if the cache is still valid. For example, timestamping could be used to determine which definitions need to be recompiled in an editor buffer; a definition has changed if its "edit timestamp" is more recent than its "compile timestamp."

Timestamping provides satisfactory cache management if the sole concern is that only valid information is ever seen by the application program. However, this is often not the only concern when using cached information. When the application using cached data is memory-intensive, for example, it is important that the memory

occupied by the cached data be freed for garbage collection purposes when the cache is no longer valid.[2] When timestamping is used, there is no way of knowing that there are invalid objects present in the address space until an attempt is made to use the objects.

A method similar to timestamping involves marking the primitive data (from which the cache is derived) as modified. Then, when the cache is needed, a check is made to determine if the primitive information has been modified. If it has, then the cache must be invalidated. "Invalidation" is the process of marking something as invalid or "not valid"; data can be invalid, but not yet be invalidated. This approach has the same drawback as timestamping in that it provides no mechanism to automatically remove invalid objects from the virtual memory.

2.2.2 Garbage Collection

A cache management system is similar to the garbage collector used in Lisp Machines in that one of its goals is to eliminate "garbage" from the virtual address space. There are, however, some significant differences.

Conventional garbage collection uses a simple notion of what is garbage; an object is garbage if, starting from a system "root," the object cannot be reached by following pointers. A cache management system takes the notion of garbage one step further, allowing invalid or superfluous cached objects to be released to the garbage collector. Garbage now is viewed in a broader sense, denoting information that is not needed anymore, not just data that cannot be reached by following pointers (although, clearly, the latter is a subset of the former).

Some garbage collection schemes (such as Lieberman and Hewitt's [6] and Moon's [9]) take advantage of dynamic information, including how long an object

[2]Here, it is assumed that if the cache is invalid, then it is no longer useful. Later, this assumption is relaxed and a more general concept of usefulness is presented.

has been around, as an indicator of how long the object will be around in the future. This only improves the performance of the garbage collector; it does not affect what is considered to be garbage. A cache management system can take advantage of this as well as other indicators to determine what is garbage in the broader sense.

2.2.3 Resources

Resources [10] are a mechanism designed to give the programmer more control over memory allocation and deallocation than is normally used, in order to improve memory utilization. Normally, in a Lisp Machine environment, the programmer need not be concerned with memory management because whatever is no longer accessible (by following pointers from the system root) is automatically garbage collected. Resources are appropriate when either very large objects are used and discarded at a moderately high rate, or when moderately large objects are used and discarded at a very high rate.

A resource is a set of related cached objects that are intended to be reused instead of recreated whenever a similar object is needed. For example, there are resources for the various types of windows and menus commonly used by the system. Whenever a "system menu" is needed, instead of always creating a new one,

```
(using-resource (x system-menu)
  ...)
```

is executed. If there is a free system menu in the resource, it is fetched and bound to x (within the body of the **using-resource**); otherwise, a new system menu is constructed using the *constructor* function, one of the parameters supplied when defining the resource. It is transparent to the application program whether a new system menu was created or an old cached one was used.

This is almost what is desired for a software cache management system. Re-

sources can check validity, one of the goals of SCMS, by using functions supplied when the resource is defined. However, resources still need to be explicitly purged; there is no implicit mechanism that purges superfluous objects in resources. Resources suffer from the same drawback as timestamps — invalid and superfluous objects are not purged until the next attempt is made to use them.

2.2.4 Virtual Memory Paging

In virtual memory architectures, paging mechanisms are used to control which sections or pages of virtual memory are present in main memory. Paging mechanisms address the same two tasks that a software cache management system must address: validity and storage management.

This domain is special in that both definition transparency and use transparency are achievable. This is possible because paging mechanisms only attempt to address the two tasks in a limited sense. The validity of a page is generally determined by a "dirty bit," indicating whether the page has been modified since being brought into main memory. This determination of validity is application independent — it does not matter what kind of information resides on the page. However, paging mechanisms do not address the issue of validity in the broader sense; that is, whether or not an object is valid with respect to application specific indicators. Storage management is also easier in this domain than in the general case because objects never have to be recreated; they are always available, intact, in secondary memory. As with SCMS, paging mechanisms need to be able to determine whether an object (page) is superfluous. In the case of paging mechanisms, superfluousness normally depends on whether the page is valid (which affects whether or not the page will have to be copied to secondary memory) and how recently the page has been used.

2.2.5 Explicit Storage Control

In some object management systems (such as CORLL [12]), there is a notion that it may be too expensive to keep some data readily available. However, in these systems, it is either the case that overly expensive (superfluous) objects must still be explicitly deleted or that the only effect of an object being superfluous is that the object is more likely to be paged out of main memory, not out of virtual memory.

Timestamping, the first system presented, addresses the problem of ensuring that invalid cached information is actually invalidated and not accidentally used, but fails to address the memory management problem. Garbage collection addresses the memory management problem to a limited degree, but has a restricted notion of what memory can be reclaimed. The resources mechanism addresses both the validation problem and the memory management problem, but freeing memory requires explicit action by the application program. Virtual memory paging addresses both problems successfully, but in a limited domain. Finally, explicit storage control fails in both respects: it does not help to solve the validation problem and memory management is accomplished explicitly as with resources rather than implicitly.

2.3 Pragmatic Approach to the Cache Management Problem: SCMS

This section describes SCMS, a software cache management system which addresses both the validity and memory management problems. In designing SCMS, a pragmatic approach was taken toward both definition transparency and use transparency.

Instead of complete definition transparency, the programmer must define an interface to SCMS that enables SCMS to make the decisions and perform the ac-

tions described in Section 2.1.1. The programmer provides cost functions, which can be defaulted, to enable SCMS to decide the relative value of cached data and determine the validity of the data. Additionally, in the interface, "action" functions are provided that enable SCMS to remove cached objects from virtual memory and recreate them when necessary.

A compromise was also made with use transparency. Special syntax must be used whenever the application program wants to use cached information. This approach incurs a small performance penalty whenever cached information is used, but incurs no such penalty when "normal" information is used.

There are three basic pieces comprising SCMS: caches, handles, and cache managers. Caches are used to retain information about individual cached objects; handles are used by the application program to keep pointers to the individual caches; and cache managers (CMs) are used to keep track of all of the caches. Finally, a list is maintained to keep track of all of the CMs. There can be many handles assigned to a specific cache (although there is normally only one), many caches assigned to a specific CM, and many CMs coexisting for use by a specific application program. Each handle is assigned to exactly one cache and each cache is assigned to exactly one CM, where each CM is created in order to track related cached objects.

There are no explicit pointers (that the application program can access) linking handles to caches. Instead, all of the links from handles to caches are contained in the CM. This means that, given a handle, the CM is needed in order to determine the handle's associated cache. This approach is taken primarily to ensure that the CM can always determine which caches are in use by the application program and to simplify the use of SCMS in distributed systems.

The handle to cache mapping scheme is similar to a hash table mechanism,

where an applications has no direct pointers between *keys* and *values*; rather, to determine the *value* associated with a *key*, a lookup must be performed in the hash table. In fact, one of the implementations chosen for CMs is hash tables, where the *keys* are handles and the *values* are caches. Other implementations include association lists and property lists.

For example, to demonstrate the role of handles, caches, and cache managers, consider the VLSI schematic bitmap application discussed in Chapter 1. First, an interface must be defined with the **defcachemanager** macro. This involves designing a small number of parameters that provide SCMS application specific information about bitmaps. A fragment of the interface appears as follows:

```
(defcachemanager bitmap (circuit-part)
  :constructor create-bitmap
  :validity-checker bitmap-validity-checker
  ...)
```

Executing the above code creates a CM named *bitmap* with a single input parameter, *circuit-part*. The parameters to the CM (in this case, only *circuit-part*) are arguments to the constructor function (**create-bitmap**). Then, to create a cached bitmap of a capacitor, for example,

```
(setq capacitor-handle (allocate-handle bitmap :capacitor))
```

is executed. The call to **allocate-handle** returns a handle to the cached bitmap of the capacitor (to which *capacitor-handle* then is bound) and creates a cache for the bitmap of a capacitor by calling the constructor function. Then, to use the cached bitmap (to draw it on the screen, for example), the **using-cache** special form is used:

```
(using-cache (x capacitor-handle)
  (draw-bitmap-on-screen x))
```

Within the body of the **using-cache** special form, the variable x is bound to the

"value" of the cache, that is, the actual bitmap array.[3] The application must be careful not to keep "permament" pointers to the cached object, that is, pointers that remain after **using-cache** is exited. This would circumvent the ability of SCMS to control the use of cached objects by possibly creating multiple, different copies of a cached object.

Notice that the application program never has direct access to the cache; **using-cache** allows the application program access to the "value" of the cache, when given a handle. The value of the cache (also referred to as the *object* associated with the cache) is different than the cache; this distinction will be clarified in Section 2.3.1.

2.3.1 The Cache

SCMS is implemented in Flavors, the Lisp Machine's object-oriented programming paradigm. Each cached object is represented by an instance of a flavor, where a flavor is a hierarchical data type.

In this thesis, the term OBJECT refers to the cached object (what the application program sees) and the term CACHE refers to the entire flavor instance that is used to keep track of OBJECT. That is, the OBJECT is part of the CACHE. Although different cache flavors can have different associated "instance variables" (parts), there are six instance variables essential for SCMS operation:

Object is the actual cached object. When the "value" of the cache is requested, *object* is returned.

State Block is additional information available for use by the CM's functions. This is auxiliary information specific to an individual cache, used internally only. That is, the *state block* is not returned by the CM when the value of the cache is requested. For example, for the VLSI bitmap example presented in

[3] There is an alternate syntax of the **using-cache** special form that both allocates a handle and allows use of the cached value. The simpler form is presented here to demonstrate the differences between a *handle* and a *cache* more clearly.

Chapter 1 and later presented in more detail in Section 3.4.1, the *state block* might contain the symbol **:transistor** in order to distinguish this cache from the one for resistors, while *object* would be the actual **bitmap**.

Use Timestamp is the time that this cache was last used. This is primarily in case the utility of the cache depends on how recently it has been used. It is updated every time the application program requests the cached object.

Scavenge Timestamp is the time that this cache was last scavenged. A detailed explanation of what scavenging a cache means is presented in Section 2.3.2.

Locks keep track of "who" (which handle or handles) is currently using this cache, that is, who has the cache "locked." This information is necessary for the operation of the CM, in order to determine whether it is safe to modify the cache.

Status indicates the validity of *object*. There are four legal values for *status*: **VALID**, **INVALID**, **ABSENT**, and **DESTROYED**. The meaning of each of these four values for *status* is presented in Section 2.3.2.

2.3.2 The Cache Manager

A CM is defined by an interface, which is a set of parameters, to the application program. The parameters include:

constructor: A function — $f(CACHE, P_1, P_2, \ldots, P_n)$ — that creates the cached object, where $P_1, P_2, \ldots, P_n$ are parameters supplied as arguments to **allocate-handle**.

destroyer: A function — $f(CACHE, OBJECT)$ — that "destroys" the object.

destruction cost: A function — $f(CACHE, OBJECT)$ — that returns an estimate of how expensive it is to destroy the object (execute the *destroyer* function).

maintenance cost: A function — $f(CACHE, OBJECT)$ — that returns an estimate of how expensive it is to maintain the object.

recreator: A function — $f(CACHE, OBJECT)$ — that "recreates" the object, that is the *recreator* is input the destroyed object and returns a recreated, valid object.

recreation cost: A function — $f(CACHE, OBJECT)$ — that returns an estimate of how expensive it is to recreate the object (execute the *recreator* function).

recreate immediately-p: Either **t** or **nil**. Determines whether or not caches are recreated immediately after they are destroyed.

validity checker: A predicate function — $f(CACHE, OBJECT) \rightarrow \{\mathbf{t}, \mathbf{nil}\}$ — that indicates the validity of OBJECT.

expected payoff: A function — $f(CACHE, OBJECT)$ — that returns an estimate of the probability that OBJECT will be needed again.

initial status: The initial status of newly constructed caches. Normally, it is **VALID**, but this parameter can override this default value by setting *status* to **INVALID** or **DESTROYED**.

manager flavor: Determines the flavor used for the CM implementation. This affects several things including the type of object table used to track the cached objects (hash table, association list, tree, etc.) and any special information the CM must track (such as least recently used indicators). The default type of cache manager is a simple hash table cache manager.

cache flavor: Determines the flavor used for the cache implementation. Different cache flavors can have different default behavior and different associated methods (functions) that are used for the cost and "action" (constructor, destroyer, and recreator) functions.

scavenge-p: Either **t** or **nil**. Determines whether or not the cache manager operates with a scavenger.

The parameters of a cache manager can be modified by reexecuting the **defcache-manager** macro with the same name specified, but with different parameters. For each of the parameters for the application, where a function is called for, either a *form* or a *symbol* is acceptable. If a form is used, a function with the form as the body is created and used; otherwise, the symbol is interpreted as a function name to be used.

To allow maximum flexibility, there are four possible values for the status of a cache. When a cache is first constructed, *status* = **ABSENT**. This value indicates that the *object* instance variable of the cache is meaningless, that is, it has not yet been initialized. After *object* is initialized, the value of *status* cycles among the three other values: **VALID**, **INVALID**, and **DESTROYED**. The *constructor* function first

creates the object and *status* is set to **VALID**. The condition *status* = **VALID** indicates that the cached object was determined to be valid the last time a check was made.

When the cached object is determined to be invalid, *status* is changed to **INVALID**, but *object* is *not* changed. The condition *status* = **INVALID** indicates that the cached object was determined to be invalid the last time a check was made, but that *object* has not yet been affected.

Finally, *object* is set to *destroyer*(*object*) and *status* is set to **DESTROYED**. This approach is taken (instead of simply setting *object* to **nil**) because even though a cached object is no longer valid, it is not necessarily useless. For example, if large arrays are being cached, an array may become invalid if a single element is no longer correct. It would be wasteful to completely delete the array and then recreate it because of one invalid element. Instead, in this example, the "destroyed" *object* could be the same as the invalid *object*. When the cache is later recreated, the *recreator* function corrects the one invalid element of the array and *status* is reset to **VALID**. This is how the "modify" operation can be simulated with SCMS. The statuses **INVALID** and **DESTROYED** are distinguished because modification is not always appropriate; sometimes the *object* should be set to **nil** in order to better utilize available memory.

Figure 2.1 illustrates what happens when an application requests a cached object. Normally, when a cache is first created and initialized (with the **allocate-handle** command), the *status* is set to **VALID** and the *object* is constructed; however, the *initial status* parameter can be used to override this default setting of **VALID**.

When a cached object is requested (with the **using-cache** command), first a check is made to see if *status* = **VALID**. If they are not equal, then *status* must be either **INVALID** or **DESTROYED**. If it is the former, then *object* must be destroyed

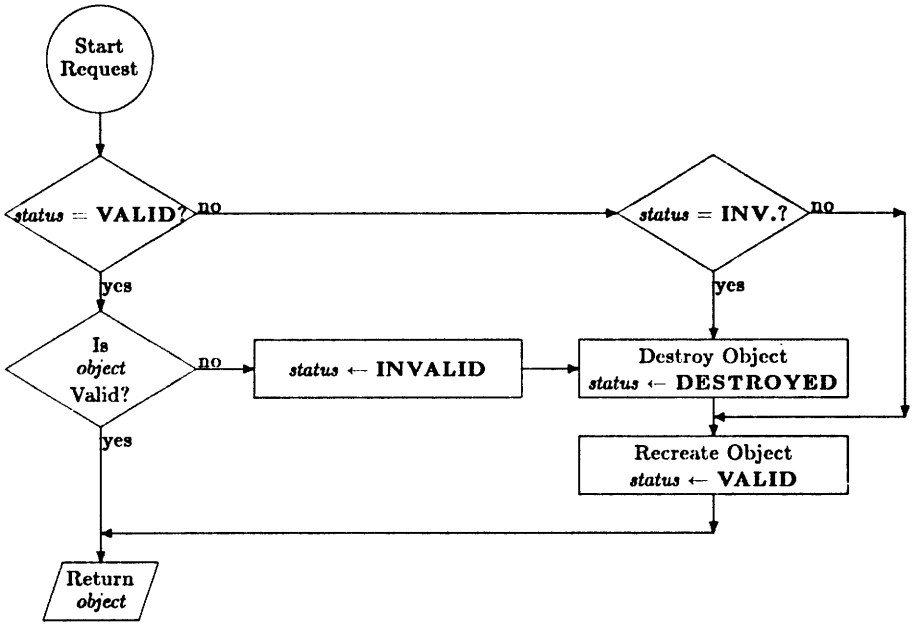

Figure 2.1: Request for Cached Object

before it is recreated. Otherwise, it is simply recreated (and then asserted to be valid). The cycle is: **VALID**, **INVALID**, **DESTROYED**, then back to **VALID**. The only time that the *status* can be **ABSENT** is when the cache is first created before it is fully initialized.

There are two different operating modes for SCMS: *with* scavenging and *without* scavenging.[4] The mode is specified by the *scavenge-p* parameter in the SCMS interface. The scavenger operates analogously to the scavenger of the Lisp Machine's garbage collector, as an asynchronous background task which continually searches for superfluous and invalid cached information. The scavenger operates as a low priority process so that it does not interfere with other processes. It takes the appropriate action on each cached object it "scavenges," where the appropriate action is determined by the CM interface to SCMS.

[4]Section 2.4 presents examples of the situations and types of applications where the scavenger *is* used and where it *is not* used.

Note that when the CM is operating without the scavenger, the *status* is normally always **VALID** whenever a request is made. This is because a request always leaves the *object* in a valid state and nothing causes this condition to change. The only times that the initial test for *status* = **VALID** fails are when the application program asserts that the cache is no longer valid or when the *initial status* parameter is used. Asserting that a cache is not valid can be done with the **assert-status** command to eliminate the need for the CM to make the check for validity or if the programmer wishes to explicitly free memory.

The sequence that is followed when a request is made for cached data and the CM is operating *with* a scavenger is identical to the sequence for when the CM is operating *without* a scavenger. However, when there is a scavenger, it is possible for *status* to be **INVALID** or **DESTROYED** without an assertion being made from the application program because the CM's scavenger can modify the state of the cache.

Figure 2.2 illustrates the sequence taken when a cached object is "scavenged." First, a check is made to see if *status* = **VALID**. If they are equal, then the cache was determined to be valid the last time a check was made and it must be verified that the cache is still valid. If the cache is no longer valid, then *status* is set to **INVALID**. Next, if the cache has not been destroyed, then it needs to be determined whether the cache is cost-effective. If it is not, then the cache is destroyed. Finally, if *recreate immediately-p* is t, then the cache is recreated.

2.3.3 Utility

SCMS' scavenger makes decisions based on concepts of *utility* and *cost-effectiveness*; a cache is cost-effective if its utility (U) is greater than the threshold value, CE_T. (The default value for CE_T is 0.) There are four parameters that contribute to U:

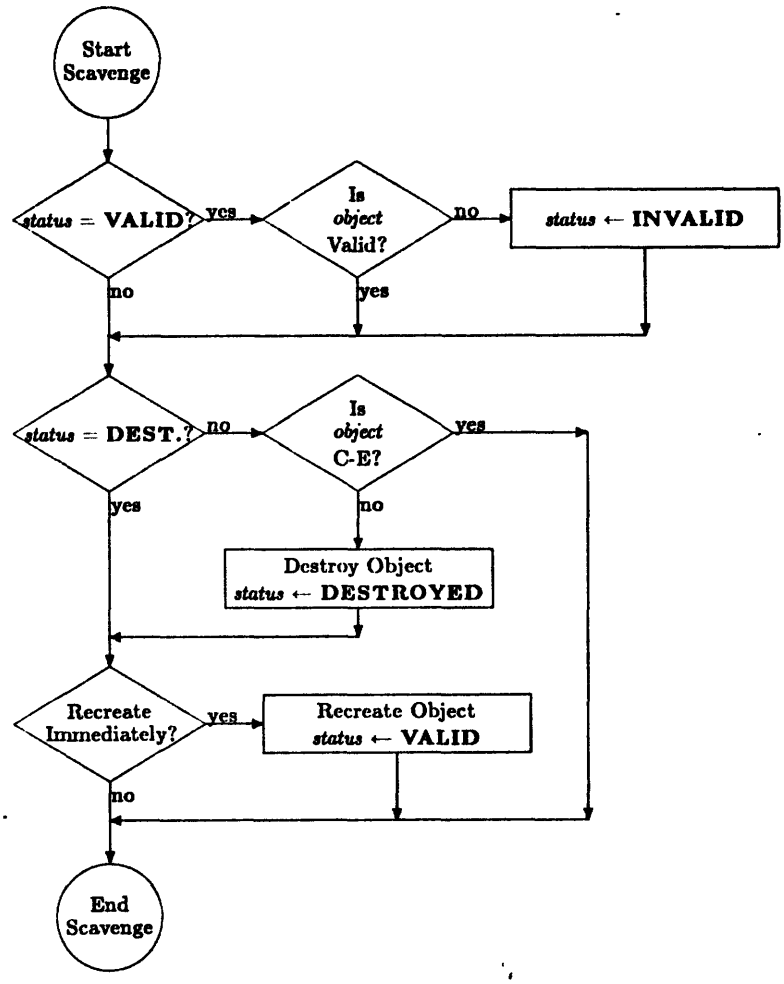

Figure 2.2: The Scavenge Cycle

expected payoff (E), recreation cost (R), destruction cost (D), and maintenance cost (M). U is determined by the following formula:

$$U = ER + D - M.$$

The four parameters, E, R, D, and M are interpreted as follows. E is a metric for how useful *object* will be in the future in its current form; that is, if *object* is valid, then E measures the expected future need for *object*. Likewise, if *object* is invalid, then E measures the expected future need of the invalid version of *object*. E is the probability that *object* will be used again ($0 \leq E \leq 1$).[5] R is a metric

[5]This range of values for E is not enforced by SCMS; however, the derivation of the formula for U

for how expensive it is to recreate *object* updating *status* from **DESTROYED** to **VALID**. D is a metric for how expensive it is to destroy *object*, that is, change the *status* of *object* from **VALID** or **INVALID** to **DESTROYED**. Finally, M is a metric for how expensive it is to maintain *object* in its current form. For the three parameters, R, D, and M, the measure of expense is in "utils," an arbitrary unit of utility borrowed from economics; the metric E is unitless.

The above formula ($U = ER + D - M$) is derived from the following boundary conditions:

$$U = \begin{cases} D - M, & \text{if } E = 0; \\ R + D - M, & \text{if } E = 1. \end{cases}$$

If the cache will definitely not be used again ($E = 0$), then the cache is cost-effective if it is more expensive to destroy than it is to maintain. Alternately, if the cache will definitely be used again ($E = 1$), then the cost of recreating the cache, R, must be added to the value because the cache *will* need to be recreated if it is destroyed. Combining the two boundary conditions and making the non-boundary conditions linear yields the formula for U.

Since the cost functions can use global variables in their calculations to determine total value, global optimizations can be performed over either all or a subset of the cache managers. This makes it possible, for example, to determine the least recently used cache over a set of cache managers.

2.4 Types of Domains

Although SCMS can be used for several different purposes, there are two primary reasons for using SCMS. Applications for SCMS tend to fall under one of both of these categories: validation and garbage collection.

---

is based on viewing E as a probability.

```
(defun make-object-function (CACHE OBJECT)
  (setf (state-block) global-phase)
  (object-from-phase global-phase))

(defcachemanager simple-validation-example ()
  :constructor make-object-function
  :validity-checker (= (state-block) global-phase)
  :recreator make-object-function
  :scavenge-p nil)
```

Figure 2.3: Simple Validation Example

2.4.1 Validation Applications

In validation applications, the primary reason that SCMS is being used is to ensure that only valid cached information is ever seen by the application program. This is useful if the cached objects are not very large and therefore garbage collection of them is not a real concern. In validation applications, there is generally no need to run SCMS with the scavenger because it is not very important to eliminate the invalid objects at the time they becomes invalid; it is only important that the invalid objects are not retrieved when the objects are needed.

A sample validation application example is shown in Figure 2.3. In this example, the cached object is a function of *global-phase*. When the object is retrieved, if the *global-phase* has changed, then the cached object is no longer valid and must therefore be recreated. The *state block* contains the *global-phase* at the time the object was created. It is retained in order to determine if it has changed over time.

The advantage of operating without the scavenger is that there is no unnecessary overhead associated with this cache manager due to resources required for the scavenger process' operation. There are situations, however, where it is advantageous to operate a validation application cache manager with a scavenger. Consider

```
(defcachemanager expensive-destroy-validation-example ()
  :constructor (progn (setf (state-block) global-phase))
                      (object-from-phase global-phase))
  :validity-checker (= (state-block) global-phase)
  :recreator (progn (setf (state-block) global-phase)
                    (object-from-phase global-phase))
  :destroyer (save-object-to-file object)
  :destruction-cost -1
  :recreation-cost 2
  :maintenance-cost 0
  :expected-payoff (if (send object :valid-p)
                       1.    ;;one if still valid
                       0)))  ;;zero otherwise
```

Figure 2.4: Validation Application with Scavenging

the case where the destroy operation is time-consuming (perhaps because the object must be saved to a file) and where it is not acceptable to have to wait for the destroy operation every time an invalid object is demanded. Once the object is no longer valid, it should be saved to the file as soon as the Lisp Machine is idle so that it is not necessary to wait for this saving to be performed when the object is requested the next time. That is, it is desired that, whenever possible, the saving take place in the background, rather than in the foreground.

This effect can be achieved in several ways, one of which is by assigning the cache a negative destruction cost to make the total value, $ER + D - M$, negative when the cached object is no longer valid. The parallel application to the previous one is shown in Figure 2.4.

Notice that in this example, the cache manager is run with a scavenger. An analysis of the cost functions reveals that if the object is valid, then

$$\begin{aligned} U &= ER + D - M \\ &= (1) * (2) + (-1) - 0 \\ &= 1 \\ &> 0. \end{aligned}$$

The object will therefore not be reclaimed if it is valid. Once it is no longer valid, however, then

$$\begin{aligned} U &= ER + D - M \\ &= (0) * (2) + (-1) - 0 \\ &= -1 \\ &\not> 0 \end{aligned}$$

and the object will be destroyed (saved to file). Since *recreate immediately-p* is nil, the object will not be recreated until it is once again demanded. Changing this parameter to t would have the effect of immediately recreating the object after the destroy operation.

2.4.2 Garbage Collection Applications

In garbage collection (GC) applications, the primary reason that the SCMS is being used is to keep overly expensive objects out of the address space to make room for more important objects. GC applications always need to be run with a scavenger. The scavenger determines whether or not a cached object is worth maintaining. In this type of application, it is possible for a cached object to be purged even though it is still valid. This happens when the cache is determined not to be cost-effective. Using the formula for utility (assuming CE_T is 0), a cached object is reclaimed when

$$U = ER + D - M \not> 0.$$

Rearranging, an object is reclaimed when

$$M \geq ER + D.$$

```
(defcachemanager GC-example (seed)
  :constructor (progn (setf (state-block) seed)
                      (expensive-to-maintain-array-from-seed seed))
  :recreator (expensive-to-maintain-array-from-seed (state-block))
  :destruction-cost 0
  :recreation-cost 1
  :maintenance-cost*
     (* (array-element-size object)   ;; bits per element
        (array-dimension-n 1 object)  ;; number of columns
        (array-dimension-n 2 object)) ;; number of rows
  :expected-payoff (some-indicator-of-future-use cache))
```

Figure 2.5: Garbage Collection Application

Figure 2.5 shows a sample GC application. In this example, objects are always valid and hence invalidity is not a reason for reclamation. The *validity checker* option is not present because it defaults to t (the object is always valid). Notice that the parameter :*maintenance-cost** is used instead of :*maintenance-cost*. This is specified in order to cache the results of evaluating the *maintenance cost* function. It is evaluated only once and then cached because M does not change for a given cached object and M is relatively expensive to calculate repeatedly. M is not reevaluated unless the object is recreated.

Chapter 3

Applications

The chapter presents four applications areas for SCMS: a clock display, functional and semi-functional programming, relational database management systems, and Schema.

3.1 Clock

A simple application, presented to demonstrate the interaction of the various parameters of the CM, is a clock display. Consider a clock where the displayed time needs to be updated every minute. Figure 3.1 shows the cache manager definition for this application. In this application, there is one parameter to the CM: *window*. This is the window where the clock is to be displayed. When the cache is first constructed, the window is put in the state block in order to keep track of this information. Also, the current date and time is displayed on this window (with the call to **print-current-time**) after the window is first cleared. Finally, the constructor function returns *minute*, an integer representing the current minute. If the time is 3:48 P.M., for example, then *minute* is 48. The CM sets *object* to 48, the value returned by the constructor function.

Every time that this cache is scavenged, a check is made to see if the cache is

```
(defcachemanager clock (window)
  :constructor (multiple-value-bind (ignore minute)
                   (get-time)
                 (setf (state-block) window)
                 (send window :clear-window)
                 (print-current-time window)
                 minute)
  :validity-checker (multiple-value-bind (ignore minute)
                        (get-time)
                      (= minute object))
  :recreation-cost 1
  :recreate-immediately-p t
  :recreator (multiple-value-bind (ignore minute)
                 (get-time)
               (send (state-block) :clear-window)
               (print-current-time (state-block))
               minute))
```

Figure 3.1: Clock Cache Manager

still valid. The cache will remain valid until 3:49 P.M., when the check

```
(= minute object)
```

will return nil indicating that the cached minute, 48, is invalid. At this time, the object will be destroyed. Since no destruction function is specified, a default function is used that simply returns nil. At this point, the *status* is **DESTROYED** and the *object* is nil. Since *recreate immediately-p* is t, the cached minute is immediately recreated. The recreation function is identical to the construction function except that the window is extracted from the state block instead of from input parameters. Hence, the time is displayed and *object* is updated to 49, the new current minute. Also, *status* is updated to **VALID** once again.

Notice that *object* does is not destroyed simply because the old *object* is no longer valid; the cause for destruction and recreation can be seen by analyzing

$ER + D - M$, the utility function. While *object* is valid, the expected payoff is 1 (the default); R is 1; both the destruction and maintenance costs are 0 (the default):

$$\begin{aligned} U &= ER + D - M \\ &= (1) * (1) + 0 - 0 \\ &= 1 \\ &> 0. \end{aligned}$$

Since the utility is positive, the cache is not changed. Once *object* is no longer valid (the minute changes), then the expected payoff changes to 0:

$$\begin{aligned} U &= ER + D - M \\ &= (0) * (1) + 0 - 0 \\ &= 0 \\ &\not> 0. \end{aligned}$$

Now the utility is no longer positive and the cache is destroyed, and then recreated because *recreate immediately-p* is t. due to the negative recreation cost.

The clock application is an unusual example for SCMS because the "value" of the cache is never needed by the application program. A clock is instantiated by creating a cache, that is,

```
(allocate-handle clock *a-window*)
```

is executed. However, **using-cache** is not needed because the value of the current minute is not important except to the CM's scavenger. After the cache is created, the clock remains active and is updated asynchronously every minute by the CM's scavenger without any additional intervention by the application program. Several clocks can operate simultaneously simply by creating several handles.

3.2 Functional and Semi-Functional Programming

In functional programming languages, such as Backus' FP [2,3] and McCarthy's "pure" Lisp 1.5 [7], a function always returns the same results when given the same inputs. This is because functions have no side-effects and there is no global state.

Consider an arbitrary function, **fp-function**, in a functional language. For simplicitly, it is assumed in the discussion that follows that **fp-function** takes one argument and returns one value. The example can easily be extended to multi-argument functions that return more than one value. If **fp-function** is a relatively complex function to compute, then it may be advantageous to cache **fp-function**'s operation by saving the results of calls to **fp-function** and using these cached results if **fp-function** is called later with the same input. If **fp-function** is rarely called with the identical input twice, then caching is most likely not appropriate for this function. However, if **fp-function** is frequently called with a limited number of inputs, then caching **fp-function**'s results may prove advantageous. If **fp-function** is being cached, then when **fp-function** is called with an argument x, first a check is made to see if the value **fp-function**(x) is already cached. If it is, then the cached result is returned, eliminating the need to recompute it. If **fp-function**(x) is not already cached, then **fp-function** is called and the result is computed and then cached for future use before it is returned.

This behavior is easily achieved, with or without using SCMS as the caching interface. Abelson and Sussman [1, p. 218], for example, present a programming technique, "memoization," whereby results of previous function calls are saved in a table in case they are later needed. They use **fib** as an example, a function that calculates Fibonacci numbers. This effect is also achieved in Prolog, as demonstrated by O'Keefe [11], where a **cache** predicate unifies with previously computed results

and returns the cached result.

There are two slight modifications to this scenario for which SCMS becomes very helpful: when the cached results are expensive to maintain and when **fp-function** is only "semi-functional." These two modifications parallel the two primary domains for SCMS: garbage collection and validation applications.

If the cached results of **fp-function** are expensive to maintain, for example, if they are memory-intensive, then it is not practical to keep cached all of the results of the different calls to **fp-function**, even though there may only be a limited number. SCMS can be used to determine which cached results of calls to **fp-function** are cost-effective and which are not.

For this application, cache handles are not needed; rather, a simpler CM lookup mechanism is needed to determine if **fp-function**, applied to a given input, is present in the cache. The CM's lookup mechanism keys off of the possible inputs to **fp-function**. In order to determine which cached results are cost-effective, the CM uses a least recently used (LRU) algorithm[1] to determine which caches are likely to be used again. Whenever a cache is used, the CM updates the LRU ordering of the caches. A CM with these two special characteristics — no handles and LRU tracking – is obtained with the *manager flavor* parameter in the CM interface which is set to *LRU-no-handle-cache-manager.*

Figure 3.2 presents a CM interface for this example. The CM reclaims objects when U is no longer positive, which occurs when $E \leq 10$.[2] **LRU-position** is a function that determines E based on how recently the cache has been used. In this example, for the five most recently used caches, $E = 20$; for the sixth most recently

[1]In the LRU algorithm used, whenever a cache is requested, it is moved to the head of a list (the *LRU-list*) of the ordering of cache usage. The least recently used cache then propogates to the tail of the list.

[2]The range of $0 \leq E \leq 20$ is used in this example rather than $0 \leq E \leq 1$ for performance reasons; it is more efficient to multiply integers than it is to multiply floating point numbers.

```
(defun LRU-position (cache number-of-highest highest step)
  (declare (special LRU-list))
  (let ((position (find-position-in-list cache LRU-list)))
    (cond ((< position number-of-ones) highest)
          (t (max 0 (- highest
                       (* step (- position number-of-ones -1))))))))

(defcachemanager fp-function-CM (input)
  :constructor (progn (setf (state-block input))
                      (fp-function input))
  :recreator (fp-function (state-block))
  :maintenance-cost 10.
  :recreation-cost 1.
  :destruction-cost 0.
  :expected-payoff (LRU-position CACHE 5. 20. 1.)
  :manager-flavor LRU-no-handle-cache-manager)
```

Figure 3.2: Functional Programming with GC Cache Manager

used cache, $E = 19$; for the seventh, $E = 18$; and so on until for the twenty-fifth and all older caches, $E = 0$. Clearly, more complex cost functions could be used if different cached results occupied significantly different amounts of memory, or if, for example, the complexity of calculating **fp-function** was dependent on the magnitude of the input. Then, when a call to **fp-function** is made with input x,

```
(using-cache-no-handle (fp-function-CM x)
  ...)
```

is implicitly executed in order to obtain **fp-function** at x. **using-cache-no-handle** is the same as **using-cache** except that the CM used must be explicitly specified as an argument to **using-cache-no-handle** because there is no handle available to determine the correct CM.

Notice that in this example, no validity checker is necessary. This is because, in functional language, there is no state to affect the validity of the cached results.

In a non-functional language (such as Zetalisp), many functions are nevertheless functional, that is, the value or values returned are *only* dependent on the input. Examples include **plus**, **car**, **setq**, **cond**, and **factorial**. There are also functions that are "semi-functional," that is, the output is dependent on the inputs and a limited, known, part of the state.

For both of these cases — semi-functional functions and functional functions within a non-functional language — there could be a declaration to advise the compiler that a function has a CM associated with it. For example, in Zetalisp, **fp-function** could be defined as follows:

```
(defun fp-function (input)
  (declare (functional fp-function-CM))
  ...)
```

Then, the *fp-function-CM* cache manager would be associated with **fp-function**. The difference when **fp-function** is only semi-functional is that there would need to be a *validity checker* parameter for the CM interface to determine if the state has changed sufficiently to make the cache invalid.

3.3 Relational Database Management Systems

Relational database management systems offer three opportunities to use SCMS: secondary indices, view materializations, and inter-query optimization.

3.3.1 Secondary Indices

In a relational database, secondary indices give ready access to those tuples of a relation that are associated with values of the indexed attributes. The sole purpose of secondary indices is to increase efficiency of queries involving indexed fields. Thus, if an employee relation is indexed on AGE, all employees over 40 can be located

using the AGE index without searching the entire table. Such indices are otherwise redundant. When a retrieval is performed into the database, if these secondary or cached indices are present, then they can be used to speed up the retrieval. If they are not present, then the retrieval can still be performed, only more slowly.

SCMS can be used to maintain indices and destroy them when available memory is running low or when maintenance costs become high, as when lots of updates occur to the relations that the indices cover. In this way, a somewhat self-organizing storage implementation level can be maintained by dynamically changing the fields that are indexed to those that will have the biggest impact on performance.

3.3.2 Views

In relational databases, a *view definition* is a named, stored, database query, and a *view materialization* is the result of applying a view definition to the current database. Few or no commercial database management systems implement views as materializations. Using SCMS, view materializations could be used to cache retrievals from the database; a materialization of a view definition might be employed several times in one or more queries as long as the base tables that define the view have not been changed.

SCMS can be used to determine both validity and superfluousness of cached view materializations. When the base tables of the database are modified, then a view materialization becomes invalid and it becomes necessary to obtain a new materialization for a view definition. If a view materialization is invalid or has not been used recently, then SCMS can delete the materialization in order to reclaim memory. This allows inter-query optimization by retaining view materializations from one query to the next, without the overhead cost of retaining *all* materializations.

For example, if a relational database contains the relations "baseball lovers"

and "football lovers," then executing the query "Find people who are baseball lovers or football lovers" would cause a view to be materialized. If a new person is added to the database, then the materialized view would have to be invalidated only if that person loves baseball or football. Also, if there are no queries related to football for a "long" time, as determined by cost functions, then SCMS would purge materializations that are based on the football lovers relation. This is just a special case of the functional programming application presented in Section 3.2, where SCMS is used to maintain a cache of previously computed results of function calls.

3.3.3 Inter-Query Optimization

Texas Instruments' NLMenu system [5,13] is an example of a menu-based relational database interface that encourages a user to specify queries incrementally. A user may ask "Find ships in the Pacific within 100 miles of Hawaii," execute this query, and then incrementally add the qualification "and whose maximum cruising speed is greater than 30 knots," or modify the original query to specify 200 miles instead of 100, and execute this new query. This way of formulating a query is natural since query formulation is a negotiation process between the user, his problem, and the state of the database. It also makes inter-query optimization potentially profitable because consecutive queries tend to be similar, containing common subqueries.

SCMS can be used to cache the parse tree for a query. Since a user's query normally only changes a piece of the parse tree, the whole tree need not be re-executed. Cached subqueries are matched against subqueries of the query being executed; those in common do not have to be reexecuted. When the tables in the database are modified, then cached subqueries need to be invalidated. Like views, this is also a special case of using SCMS to maintain a cache of function calls.

3.4 Schema

Schema [15,16], a VLSI design workstation currently under development for the Lisp Machine, is a particularly appropriate application for SCMS operation because there are several areas of Schema's operation in which SCMS can be installed. Section 3.4.1 presents an interface for caching bitmaps in Schema. Section 3.4.2 presents ideas for using SCMS to control the design hierarchy in Schema. Finally, Section 3.4.3 presents ideas for implementing incremental design verification with SCMS.

3.4.1 Bitmaps

Schema uses an ad hoc caching mechanism to cache bitmaps of schematic icons in order to eliminate the need to recompute the bitmaps every time an icon is displayed. When an icon of a resistor needs to be displayed, for example, the icon is sent a *:display* message which finds a bitmap to display from one of three places, in order:

1. this particular instance of a resistor icon;
2. a cache for resistors containing bitmaps in all the sizes and orientations that have been needed so far;
3. a bitmap is created, placed in the cache for future use, and returned.

This approach is reasonable for a production quality system; however, it is not flexible enough for a development system. If the function that draws resistors is modified in order to change the graphical representation of resistors, then there is no easy way to ensure that the old, cached bitmaps are no longer used. Deleting the cache is not practical if the function is changed several times or if some other aspect of the icon display mechanism, such as scaling icons, is being modified. It might

be desirable to completely turn off caching while some part of the system is being developed. Currently, in Schema, this is possible by setting a flag that inhibits all icon caching. However, this does not purge all the caches, it only prevents the creation of new caches. Invalid cached information may still accidentally be retrieved when this flag is set to turn caching back on. Also, it is not possible to only cache some kinds of icons and not others.

With SCMS, it is possible to improve the control of which cached bitmaps are retrieved and which are considered to be invalid and need to be recreated. Reexecuting the **defcachemanager** macro with a different *validity checker* parameter allows the conditions for cache validity to change over time. When the system is finally production quality, then the *validity checker* is modified to always return **t**, indicating that all cached bitmaps are valid.

3.4.2 Design Hierarchy

In Schema, VLSI designs are specified and internally stored hierarchically. Within a designer's portfolio, there can be several projects; each project can have both sub-projects and modules; and each module can have icons, topologies, schematics, and layouts. reduce the amount of design information that resides in virtual memory, Schema operates on a demand loading basis. That is, a piece of the hierarchy is not loaded from file until it is needed. Demand loading reduces virtual memory requirements for a design significantly; however, as designs grow larger, it is still possible to exhaust available memory.

SCMS could help to solve this problem by complementing demand paging with "automatic purging." When a component of the hierarchy remains inactive, SCMS could remove the component from virtual memory, freeing up memory for new components of the hierarchy to be demand loaded. Without SCMS, in Schema,

all components of the hierarchy that have been loaded since a Schema session has begun remain indefinitely, unless they are explicitly deleted from the design.

In a typical Schema session, one section of the design is developed for awhile, set aside, and then another section is developed. SCMS could take advantage of this behavior by purging the first section, on a least recently used basis. This mechanism could be implemented by defining a "project" cache manager and a "module" cache manager, or possibly only a "component" cache manager, if there is no need to treat projects and modules separately. The destruction cost function could be based partially on whether the section to be purged has been modified; modified sections are more expensive to purge because they first have to be saved to a file.

3.4.3 Incremental Design Verification

In circuit design, design verification is a time-consuming task which involves checking that the design meets its specifications and the design rules for the VLSI technology used. It is wasteful to verify a complete design that has already been verified, simply because a slight modification has been made to the design. SCMS could be used to implement an incremental design verification mechanism by keeping track of which parts of the design have been modified since the last verification, and which design rules and specifications need to be rechecked.

For some types of design constraints, such as timing constraints for critical paths, it is not necessary to ensure that the constraint is satisfied all of the time. It is acceptable to check these constraints periodically to see if any constraints have been violated. SCMS could assist with this process by asynchronously performing incremental design verification. This means, for example, that while the Lisp Machine is idle, incremental design verification for critical paths could be performed.

Then, when it is necessary to have the constraints completely satisfied, much less additional work is required.

Chapter 4

Conclusions

SCMS successfully demonstrates that there are properties common to cached data across a wide variety of applications, and that it is possible to use a canonical caching mechanism to exploit these properties in order to improve the performance advantage of using cached data. To install SCMS in an application, a caching interface is defined detailing application dependent information related to caching. SCMS uses a relatively simple formula to determine the utility of cached information, $U = ER + D - M$; however, many different kinds of behavior were found to be obtainable using this formula. SCMS is flexible in that different applications can easily use different formulas to determine utility. However, none of the applications required a different formula because the default formula was able to properly express the desired behavior.

It would not be sensible to evaluate the concept of a software cache management system based on performance results of SCMS; to obtain the performance gains that are possible, microcode support is really needed. Microcode support could permit a fully use transparent system, a major drawback of SCMS. It would still be necessary to exhibit special care when using cached information, but special syntax would not be necessary. The scavengers used by the cache managers could also be operated

with an even lower overhead.

A respect in which SCMS currently fails is non-interference; it is possible for SCMS to prevent garbage collection of cached data that would otherwise be garbage collected. This occurs when the only pointers to the cached data are the ones used by SCMS to keep track of the data. In the current Lisp Machine garbage collection implementation, this is unavoidable. "Soft pointers" are needed to solve this problem — pointers that are like normal pointers in all respects, except that they do not prevent the garbage collector from reclaiming objects. If an object is pointed to *only* by soft pointers, then the object can be reclaimed by the garbage collector. Without soft pointers, it is difficult to evaluate SCMS' success in controlling memory usage.

A direction for future research in this area is definition transparency. A better classification scheme for types of applications needs to be developed in order to improve transparency. It is possible, for example, for all cached data in an application to always remain valid. Functional programming, presented in Section 3.2, is an example of such an application. However, it is not possible in general to determine object validity or object utility. There are application independent heuristics, such as using how recently an object has been used to indicate how likely it is be needed again. Better application independent heuristics for both validity and utility need to be developed to reduce or possibly eliminate the need to define a caching interface when using a management system such as SCMS.

A second direction for research is expanding on the idea briefly presented in Section 3.3.1 of using SCMS to implement self-organizing storage implementations. Multiple representations are used in applications such as relational databases and MACSYMA [8], a programming system for performing symbolic as well as numerical mathematical manipulations. It needs to be seen how effective a system such as

SCMS would be for maintaining several representations consistently and choosing appropriate representations dynamically, as opposed to only maintaining a primitive represenation and a single cached representation.

Finally, possibilities of using parallel processing in this domain need to be examined. Parallel processing could be used for purposes such as maintaining multiple representations and determining more effectively relative utility among caches within a cache manager.

Bibliography

[1] Abelson, Harold and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* Cambridge: The MIT Press, 1985.

[2] Backus, J. "Can Programming be Liberated from the von Neumann Style?" *Communications of the ACM*, **21** (1978), 613–641.

[3] Backus, J. "The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions." *Proceedings of Symposium on Functional Languages and Computer Architectures*, Gothenberg, 1981.

[4] Corey, Stephen M. and M. Rajinikanth. "The *Explorer* Relational Table Management System: RTMS." *Proceedings of TI-MIX*, Texas Instruments Users Group Meeting, 1985.

[5] Kolts, John. "NLMenu (Natural Language Menu) on the *Explorer.*" *Proceedings of TI-MIX*, Texas Instruments Users Group Meeting, 1985.

[6] Lieberman, Henry and Carl Hewitt. "A Real-Time Garbage Collector Based on the Lifetimes of Objects." Communications of the ACM, **26** (1983), 419–429.

[7] McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." *Communications of the ACM*, **3**, 1960, 184–195.

[8] Mathlab Group, *MACSYMA Reference Manual*, Version 10. Laboratory for Computer Science, MIT, 1983.

[9] Moon, David. "Garbage Collection in a Large Lisp System." *Proceedings of ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984, 235–246.

[10] Moon, David, Richard M. Stallman, and Daniel Weinreb. *Lisp Machine Manual.* Cambridge: MIT, June 1984.

[11] O'Keefe, Richard. "Caching Results." *PROLOG Digest* 3, No. 1, 1985.

[12] Smith, David E. "CORLL Manual: A Storage and File Management System for Knowledge Bases." Memo HPP-80-8. Heuristic Programming Project. (Working Paper). Department of Computer Science, Stanford University, November, 1980.

[13] Thompson, Craig W. "Using Menu-Based Natural Language Understanding to Avoid Problems Associated with Traditional Natural Language Interfaces to Databases." Ph.D. Dissertation, University of Texas at Austin, 1984.

[14] Thompson, Craig W., Kenneth M. Ross, Harry R. Tennant, and Richard M. Saenz. "Building Usable Menu-Based Natural Language Interfaces to Databases." *Proceedings of Ninth International Conference on Very Large Data Bases*, Florence, Italy, 1983.

[15] Zippel, Richard. "An Expert System for VLSI Design." *Proceedings of International Symposium on Computers and Systems*, Newport Beach, CA, 1983.

[16] Zippel, Richard. "Schema — An Architecture for Knowledge Based CAD." Submitted to International Conference on Computer Aided Design, 1985.