Multiple Trace Composition and Its Uses

Adam Sah
Computer Science Division
University of California, Berkeley

Multiple Trace Composition and Its Uses

Adam Sah*

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776
asah@cs.Berkeley.EDU

May 23, 1995

Abstract

Traces are code attachments to variables that cause designated blocks of code to be executed on reads or writes to the given variable. Traces have numerous uses, including rules ("on <condition> do <action>" statements), autoloading and initialization based on data access, transparent remote data access, paging and swapping, and persistence. Traces are usually limited to ad-hoc, hard-coded composition, making it difficult to place multiple traces on the same variable. Multiple traces are useful for putting a rule on a persistent variable, or several rules on a variable.

This paper presents a design for a low-level mechanism for reasoning about and configuring multiple traces on a single variable. Although the work is based on a prototype using a modified version of Tcl's `trace` command, this mechanism easily applies to any language or library capable of implementing traps. This includes systems such as data breakpoints[WLG93] in C and overloadable get() and set() methods in prototype-based object-oriented systems.

The specifics of the prototype are presented, including a facility for writing persistent Tcl scripts. The prototype requires Tcl7.x with Tk3.x or later and may be downloaded from `ftp://ginsberg.cs.berkeley.edu/pub/asah/dmt/tcl-proto.tar.gz`

*supported by a grant from the National Science Foundation #NSF-IRI9107455

1 Introduction

Variable traces are code attachments to variables. When a program reads or writes a traced variable, the code is executed. The Tcl[Ous94] `trace` command is typical of language support and will serve as the example throughout the paper, although memory access traps are commonplace in operating systems (ie. virtual memory fault handling). Traces are also described as "hooked values" in a survey paper by Gudeman[Gud93].

Few access trapping systems support the composition of multiple code attachments on a single traced variable; in other cases, programmers of the last trap must explicitly call any previously-defined trap handler as hard code. Ideally, a system supporting composition would allow handlers to be closures (anonymous, nested, lexically scoped procedures) and automatically replace the trace handler with the next one to fire for the duration of the current handler. Such composition allows the abstraction of these "resends" so that programmers can later reorder the composition, inspect the current composition (debug it), or perform other operations on the composition, from code external to any particular trace handler.

Access traps are a powerful abstraction for library and extension writers, especially in managing data structures with constraints or in gathering disparate data into a virtualized structure similar to one already in the language (ie. Tcl array variables). In this context, traces can hide arbitrary complexity (ie. caching, remote access, encryption, etc.) behind an apparently simple interface. For example, it is possible to gather local and remote data into a single "virtual array". Since caching and prefetching are shown to be identically available in this model, performance does not suffer.

The key semantic drawback is that traces cannot take an arbitrary number of parameters: the read trace takes none, and the write trace takes exactly one (the value assigned). In practice, this abstraction only models store and retrieve; if get() doesn't return what set() stored, it is likely to confuse users. Even so, there are numerous uses for composable traces, including: read-only and uninitialized variables (section 3), rules and constraints (section 4), language support for dynamic loading and paging (sections 5 and 6), paging and transparent remote data access (sections 6 and 7), and logging and persistent computation (section 8).

This paper presents a low-level design allowing composition of traces. Section 2 introduces the semantics of tracing and composition and offers a low level mechanism for composing traces. Rather than attempt to motivate the need for multiple traces on a single data variable, I will instead present a number of useful examples, any combination of which should serve as ample motivation. Sections 3-8 are devoted to these examples. Section 9 considers efficiency as compared to offering method-based access to data structures. Section 10 discusses related work and section 11 suggests future work.

2 The Semantics of Traces

Tcl's `trace` command allows variables or named slots in an array (associative array) to be trapped on reads, writes or unsets. On a trap, a named procedure is called and passed arguments corresponding to the variable name accessed and index (if an array access) and the operation performed: read, write or unset.

The semantics of the trace facility presented in this paper differ from that of Tcl's `trace` command, with the intention of offering composability. The following are the functional requirements of the new facility:

1. traces need to be given unique identifiers (instead of being named by the procedure name and access method), removing ambiguity between two traces calling the same procedure. I propose having `trace variable` return a handle, which can then be used to refer to that trace. For additional flexibility, `trace` scripts should be parameterized blocks of code (ie. closures of some kind), not names of procedures. This would all variables local to the setting procedure to be accessed by the trace code (for Tcl, substituted by value at the time the trace is created).

2. The trace handler for writes should perform the write, not the system. On writes, Tcl eagerly performs the assignment *before* the trace is fired, making it difficult to, for example, discard the write, or otherwise effect different behavior on write calls. Instead, the body of a trace should have to explicitly set the variable on writes.

3. Traces on a variable should remain active during their execution, allowing composition; the current semantics of Tcl turn off all traces on the variable while the first trace executes. Accessing the currently-traced variable triggers what I call a "resend" (after the object-oriented programming concept of calling one's parent method during the execution of the same method in the child). By default a trace should turn itself off for the duration of its own execution; without this safety measure, the most common trace handlers would have to be carefully written to manually turn themselves off, lest they go into infinite loops resending. There should also be a way to manually turn traces on and off.

4. It should be possible to specify an ordering for resending (ie. which trace fires when a given one resends), both as each trace is added and after some or all traces are already setup. In addition, traces should be taggable as "must fire first" or "must fire last", since some traces won't make much sense in other cases. For example, read-only variables' write traces probably ought to be "must fire first", since they generate an error. For example, a trace that ultimately results

in network messages will probably want to be fired last, both to improve performance should the call be unnecessary, as well as because control flow is logically moved to the remote machine. For -first and -last, only one trace can fire first and only one last. Other traces happen in between based on their currently defined order.

For the purposes of this paper, we define a new set of semantics as they might be bound into Tcl commands. A prototype of these semantics has been added to Tcl using the existing `trace` command. This prototype, while complete, is sufficiently slow as to not constitute a useful system for most applications (4ms + 12ms per trace on a 100+ MIPS DEC Alpha 21064!) Rewriting this facility in C would be straightforward, but requires modification of the Tcl core. The new trace semantics are defined as a set of Tcl commands:

1. A command `newtrace` is added to the language and like the current facility, `trace`, it takes an option selector and option-specific additional arguments.

2. So that the code to run on a trap doesn't have to be a named procedure, the `variable` option now takes a script (block of code). In this script, the strings `%name1` and `%name2` are substituted to being the name of the variable and the index into the array (or "" if scalar), respectively. `%varname` substitutes the full variable name. In the case of write traces, the string `%newval` is substituted to being the value about to be assigned. For convenience, this block of code will be called the *trace code*. The return from the trace code becomes the result of the get() or set() call that triggered the trace.

   The return of a call to `newtrace variable` is a handle (string name) of the given trace. This handle is used in future `newtrace` calls to refer to this trace. This solves the ambiguity problem Tcl currently faces when referring to a trace by its {type, script} tuple; this arises, for example, when trying to delete a trace. In the body of the trace code, `%tracehandle` is substituted to being the handle for the current trace being set.

3. In order to deal with infinite loops, within the context of a given trace, the trace itself is disabled. The trace is reenabled at the completion of the call, even if the call terminates by throwing an "error" (which might otherwise skip past this re-enabling code). For notational convenience, reads or writes to the trace variable during the execution of trace code are called resends, since they tend to trigger other traces set on this variable.

   In order to allow users to override this behavior and to generalize this control to other situations, `newtrace` allows a user to manipulate the enabled status with the `enable` and `disable` options. Each takes a trace handle or list of trace handles, and enables or disables the given traces.

4. In order to introspect and modify the current state of traces, `newtrace vinfo` returns a similar list of traces as `trace vinfo`, only the procedure names are handles. The trace code text is also present in each entry.

   newtrace offers an option, `change`, which is used to set the trace code to a new script. It takes two arguments: the trace handle of the trace to change and the code to replace it with.

   newtrace offers an option, `order`, which is used to specify the firing order of variable traces. `newtrace order` takes a trace handle and an ordering argument, as shown in figure 1.

5. Some changes are required of Tcl tracing: First, it is required that traces can be placed over entire Tcl array variables, not just on specific slots. To get around this, the prototype creates a new version of `set` that is aware of full-array traces (`set2`). Second, Tcl's trace facility performs assignment before write traces fire; to undo this, the prototype maintains the previous value explicitly. Third, Tcl turns off all traces for a given variable during trace processing. To get around this restriction, the prototype substitutes `%varname` in rules with a new, unique variable that for which it can turn tracing on and `%realname` is substituted if the user needs the textual name that the original trace was fired on. These syntax make

`-first` this trace should that is run when the trace fires.

`-last` this trace should fire last. ie. only if all other traces resend (trigger the same read or write trace) does this trace fire.

`-early` this trace should run before any others, except the `first`.

`-late` this trace should run after all others, except the `last`.

`-before <trace handles>` this trace should fire sometime before the specified trace handles (can be a single handle or a list of handles).

`-after <trace handles>` same as before, but requests after status.

Figure 1: Options to `newtrace order`. Although any combination of orderings can be made, many don't make sense. For the purposes of this description, it is sufficient to treat these as imperative statements - these are not constraints and are not maintained between modification to the ordering or presence of traces. Each argument set within the same `newtrace order` statement can be treated as a separate call. It is convenient to think of *all* variables having a beyond-last write trace set, which stores away the new value in a place accessible only to a beyond-last read trace, also set on all variables.

some of the examples in the paper differ trivially from the code in the demo. Again, the implementation is a prototype: a serious effort will require changes to the Tcl core. The syntax hacks would not be needed in an in-core implementation and would also dramatically improve performance.

Having defined the new facility, we can now demonstrate examples of its usage. Note that some of the code examples are from the demo, and some are pseudo-code; each are labeled accordingly.

3 Read-only Variables and Uninitialized Variables

Read-only variables generate an error on write attempts, modeling a specific constraint useful in debugging programs and maintaining consistency of internal library data. Write traces can be used to implement read-only variables, as shown in figure 2.

Uninitialized variables, by contrast, generate an error on read attempts unless the language is supposed to automatically initialize variables. This is useful in ensuring that programs don't depend on the quirks in one language implementation or last value in main memory. This trace should remove itself after the first write.

4 Rules

Rules are statements of the form "on <condition> do <action>" and are like `if` statements that are always watching. Rules can be implemented with traces. For example, one rule semantic might treat (eg. compile):

```
on {$a<7} do {puts "hi"}
```

as

```
newtrace variable a write {
    if {%newval < 7} {puts "hi"}
    set %varname %newval
}
```

Rules can now be composed and have their composition modified on the fly. For example, let's suppose the above example were augmented by a second rule:

```
on {$a<5} do { puts "there" }
```

We would like to know (and control) whether `[set a 3]` causes "hi" to be printed before "there" or vice versa– we need to now know which rule will fire first. Given the above interpretation of rules, this question exactly maps onto the question of which order the traces will fire in. Of course, the semantics of multiple rule composition are more complicated than this; a future paper will outline their semantics in greater detail. The demo does not include rule support.

```
proc make-read-only {var} {          # this code from the demo
   upvar $var $var
   newtrace variable $var write {
      error "can't write \"%varname\": variable is read-only"
   }
}
```

Figure 2: Implementation of read-only variables.

```
# this code from the demo
proc data-autoload {var filename} {
    upvar $var $var             # pass arg 1 by reference
    set readhandle  [newtrace variable $var\(_) read  ""]
    set writehandle [newtrace variable $var\(_) write ""]
    newtrace change $readhandle "
 newtrace vdelete $readhandle $writehandle
        puts {autoloading $filename...}
        uplevel 2 source $filename
        return \[uplevel 1 set2 %realname\]
    "
    newtrace change $writehandle "
 newtrace vdelete $readhandle $writehandle
        puts {autoloading $filename...}
        uplevel 2 source $filename
        return \[uplevel 1 set2 %realname %newval\]
    "
}
```

Figure 3: An implementation of data-autoload.

5 Data Autoloading

A data autoload is the dynamic loading of a library based on data access, as opposed to procedure execution, the usual method for triggering an autoload. The advantage is that if the library primarily provides a data structure (as opposed to a set of methods) or could otherwise be accessed first by a data structure access, a data autoload obviates the need for an initialization routine. Whenever either the provided data or routines are accessed, the system autoloads the module.

Figure 3 shows a trace-based implementation of data autoload. If we say `[data-autoload people people.tcl]` and `people.tcl` contains a set of `set` commands to configure a global array variable `people`, then we can tell users that a variable named `people` exists, even though it doesn't really. This provides dynamic loading for data items, obviating the need for library initialization routines (ie. `require` in Lisp and Scheme) and allowing users to pretend that all extensions have already been initialized.

6 Paging

Paging is the incremental loading and saving of data from a larger secondary store. This allows us to manage memory on a finer granularity than whole modules and to automatically discard or write-back data that won't be accessed in a while. While most operating systems support this, application-level paging is more powerful because:

- data-specific compression techniques can be employed. For example, a smart image cache can compress least recently used (LRU) images when space is needed. Compressed data can then be paged to a secondary store when all images have been compressed. Some applications can even use lossy compression, allowing 10:1 or more compression ratios.

- different types of data can be cached differently. The typical example is a data set is usually scanned sequentially by the given application, a situation best handled by using a most-recently-used (MRU) caching policy for that data (ie. turn off caching).

- smarter prefetching is possible. The application can pass precise information to the prefetch routines, because it specifies exactly what to fetch early.

We can extend data autoloading to support paging with two modifications. First, we need to call an externally provided reader/writer routine. Second, we need to flush the cache periodically and/or when memory runs low. In order to prove the concept, the prototype flushes all cached values every five seconds.

7 Remote Data Access

Transparent remote data access is similar to paging, only with the additional requirement of cache coherence among local copies. For simplicity, we model only non-shared data; efficiently implementing cache coherence in distributed shared memory systems is an active area of research.

Currently, systems such as Tk and Tcl-DP are based on an RPC-like model. In this example the people database is distributed, with parts of the database cached locally and parts stored remotely. Using composable traces, we can make this distributed storage seamlessly provided to the user. On reads and writes to the remote variable, we first check the local store, and if not present, we redirect the request over the network to the remote store.

This also demonstrates the need for composability. For example, let's now suppose that we additionally want to constrain access to the people database (eg. "the database cannot contain a member 'Smith' "). Then, we want to take advantage of this constraint on the client side, in order to save the network message, which could be arbitrarily expensive. Thus, we want to ensure that the constraint trace fires before the distributed storage call.

Furthermore, if the array were completely virtualized and there was no local cache of the people database, then it would probably be an error to set traces to fire after the remote access, since they would never be called. This is one example where the `-last` option might be useful.

8 Logging and Persistent Computation

Telescript[Whi94] supports agents that can survive a host failure. Such persistence requires that agents periodically log their state to disk or to a remote host. Logging can be achieved by placing traps on persistent data variables[Sat93], noting updates, and removing the traps after the first update. At the end of a transaction (a unit of atomic, persistent work), we could copy the state of all modified data items to a disk or remote host, restoring the original traps. Transactions are required for reasonable performance; the alternative is to log every single data store to a persistent variable, which is extraordinarily expensive.

Here is an example interface:

`persistent` Declare a variable (or set of variables) to be persistent, with certain backing store parameters (such as which storage manager to use, the variable's name in the storage manager's namespace, etc.)

`checkpoint` Checkpoint modified variable(s) listed (or all of them if none listed) to their respective persistent stores.

The implementation sets traces on all persistent variables. As each persistent variable is modified, the modification is noted as part of the transaction and the write trace disabled until end of transaction (checkpoint). At each checkpoint, the current values of each modified persistent variable is written back to permanent storage.

```
# declare that the variable x be made recoverable, that it's name in
# the persistent storage manager's world is "somename" and to use the
# storage manager named "wish #2" which is a Tcl interpreter.
persistent x -name "somename" -type "Tcl"      -place "wish #2"
persistent z -name "x" -type "pg95" \
             -place "eden.cs.berkeley.edu 5432 MYDB PERSIST"

# change the value of 'x' to 1.
# It is *not* recoverable until checkpointed.
set x 1

# sends the value of 'x' to 'wish #2'.
checkpoint x

# modify 'y'.
set y 1

# saves the value of 'y' in postgres ('x' wasn't modified).
checkpoint
```

Figure 4: Persistent computation example. Note that only Tcl and Postgres95 (pg95) bindings have been written. In the Postgres95 binding, the place refers to a host, a port number, a database, and a specific DBMS table, in that order.

```
# this code from the demo...
persistent ip1 -name stored_ip1 -place fred1 -type tcl
persistent ip1 -name stored_ip2 -place fred2 -type tcl
set ip1 1
checkpoint
```

Figure 5: Composed persistence (persistence as a service).
As with the other examples, persistence demonstrates the need for composability. In this case, the checkpoint should write `ip1` back to two places.

Reading back the persistent variable happens lazily: a read trace on the persistent variable at the time of the `persistent` command call triggers a call to the storage manager on the first read. This trace is disabled by writes that happen first.

Concurrency control can be handled by acquiring locks for each persistent variable as it is touched. Locks would be dropped at checkpoint time as per two-phase commit semantics. Deadlock would be handled by having the acquire-lock routine throw a deadlock error, which the caller can catch. If uncaught, the system could catch this error and release other locks held by this program.

Note that this doesn't perform the two phase commit needed for true atomicity in the previous example or deal with other issues typically handled by real DBMSs. It is expected that such a facility would be glued on top of a more complete object-oriented DBMS back-end, such as Exodus[Car86] or SHORE[Car94]. Relational databases would be harder to glue, since this model doesn't handle joins; the right way to model this might be to represent each desired join as a view in the RDBMS and attach the Tcl interface to that view. In addition, the prototype only supports unlocked writes to variables backed by other Tcl interpreters. Adding support for locking and secondary storage should prove straightforward given the widespread availability of multiuser storage managers such as Exodus and Postgres.

9 Efficiency Concerns and Tradeoffs

One of the advantages of procedural access methods to data ("methods") is that they allow users to create higher-level data access methods, which this technique disallows because traces only trigger on generic reads and writes to single variables (or slots in collection variables). For example, without modification, this proposal would have an array version of a `foreach` loop on an array variable take one trap per element in the table.

The real difference (and a disadvantage of this proposal, admittedly) is that the use of procedures makes it easy to pass hints and other arguments to the methods. By comparison, read/write data access gives you precisely two "methods" and one of them may take a single argument. Since that single argument can be of any type, this is as powerful as any multi-argument scheme because we can accept either an aggregated argument (`set` with a list as the assigned value) or return a closure which itself takes an argument. The real limit is in how easy the facility is to use, which neither alternative maintains.

10 Related Work

The SL5 language contains filters[Han78], which are remarkably similar to the composable traces "invented" in this paper. The filters work even points out many of the same uses mentioned here, although its model of I/O and persistence is less sophisticated (probably due to the "newness" of relational databases in 1978; object databases hadn't even been invented yet!) SL5 was also influenced by notions of failure and backtracking, which puts it in a different design space than this paper, which focuses on a more mainstream host language.

Many of the issues in using traces have been explored in the database and AI literature, which call them "rules". From our standpoint, many of the semantic issues are similar, although the context is quite different.

- Unlike DBMSs, languages are designed to be fully programmable and user extensible. DBMS rules are designed to enforce constraints[HW93], to provide simple triggering mechanisms, or to provide different views of data[Sto90]. Language use encompasses a strict superset: for example, none of the DBMS papers consider rules that modify themselves, modify other rules or add extensions to the DBMS. Likewise, DBMS rule systems research is concerned with confluence and termination[AWH92] which are extremely hard to prove for even simple programs in general purpose languages (and, of course, impossible to prove in the general case).

- Unlike AI systems, we are not trying to provide automatic reasoning about uncertain or complex situations. Language-based rules are meant to be programmed manually, where AI rule systems have typically investigated automatic resolution of rule conflicts[For81].

Some object-oriented systems allow prototypes, which are objects that differ (dynamically) from the templates used to create them (eg. Self has prototypes, ScriptX has singletons). If all objects have get() and set() methods used to access them, then the overloading of get() and set() methods is precisely equivalent to composable traces, although overloaded methods on prototypes are a more general mechanism. Overloadable getter/setters can still benefit inasmuch as this work presents a higher level interface for dynamically assigning a resend order between them.

Compared with operating system virtual memory traps, traces are an instance of user-level virtual memory support (software fault isolation (SFI) [WLAG93]): when you touch a tainted piece of memory, your program jumps somewhere else and runs some code, then (possibly) returns to the caller. The primary differences are that language-level traps are orders of magnitude faster and offer facilities for composition and introspection. Compared with current SFI work, composable traces make traps a first-class language mechanism.

Finally, there has been some recent work on composition of OS services. Examples include composable file systems such as Spring[KN93] and Interposition Agents[Jon93]. The chief difference in this work is its focus on memory accesses and their interaction with the target programming language.

11 Future Work

Future work could take (at least) four directions. First, this work presents a low-level mechanism when higher level mechanisms are probably needed. Such constructs are likely to be language-dependent since, by definition, high-level mechanisms interact tightly with existing language features. Even this proposal attempts to fake the presence of closures in Tcl, which doesn't support them.

Second, this work does not flesh out how traces work on aggregate data structures (ie. arrays). This is because collection access and memory management semantics depend a lot on the target language, which would likely affect how traces are handled. This hunch was confirmed in the design of Rush, a language similar to Tcl. Among other differences, Rush supports references and first-class collections[SBD94].

Third, it would be interesting to explore whether constraints made from traps ("rules") could be used to allow untrusted scripts to run safely ("safe execution", a la Safe-Tcl).

Finally, the demo is intentionally prototypical and still quite brittle. If features such as persistence and transparent remote data access are important, then composable traces need to be implemented in the Tcl core. Since it interacts with how variables are assigned and traces fired, such changes cannot be added as a third party extension.

12 Acknowledgements

The author wishes to thank Jon Blow for helping develop many of these ideas over the past few years and Tom Lord, Sanford Barr, Raph Levien, Brian Dennis and Allison Woodruff commented on earlier drafts of this work. Thanks to Michael Stonebraker for his many flavors of support. Thanks to David Gudeman and Ralph Griswold for pointing me to SL5. The Postgres95 update to Postgres is the work of Andrew Yu and Jolly Chen.

Thanks, of course, go to John Ousterhout, whose Tcl tool proved once again to be insanely flexible: the entire prototype and demo (1300 lines) was written in a week! In other languages, this would have required substantial hacking. Specifically, Tcl's introspection capabilities proved helpful for debugging the necessarily twisty code involved.

References

[AWH92] Alexander Aiken, Jennifer Widom, and Joseph Hellerstein. *"Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism"* Proc. ACM SIGMOD'92 Conf. on Management of Data.

[BS94] Jon Blow and Adam Sah. *"Rule Semantics and Their Use in More Precise Aliasing"* UC Berkeley Technical Report #94/59.

[Car94] Michael Carey, et. al. *"Shoring Up Persistent Applications"* Proc. ACM SIGMOD'94 Conf. on Management of Data.

[Car86] Michael Carey, et. al. *"The Architecture of the EXODUS Extensible DBMS"* 1986 Proc. Symp. on Very Large Database Systems. (VLDB'86)

[For81] C.L. Forgy. *"OPS5 User's Manual"* Dept. of Comp. Sci., Cargenie-Mellon Univ. 1980.

[Gud93] David Gudeman. *"Representing Type Information in Dynamically Typed Languages"* Univ. of Arizona Technical Report #93-27.

[Han78] Hanson, David R. *"Filters in SL5"* The Computer Journal. v.21,No.2. pp.134-43. May 1978.

[HW93] Eric Hanson and Jennifer Widom. *"An Overview of Production Rules in Database Systems"* Knowledge and Engineering Review. vol.8, no.2, pp.121-143. June 1993.

[Jon93] Michael B. Jones. *"Interposition Agents: Transparently Interposing User Code at the System Interface"* Proc. 14th Symp. on Operating System Principles (SOSP'93).

[KN93] Yousef Khalidi and Michael Nelson. *"Extensible File Systems in Spring"* Proc. 14th Symp. on Operating System Principles (SOSP'93).

[Ous94] John Ousterhout. An Introduction to Tcl and Tk. Addison-Wesley. 1994.

[SBD94] Adam Sah, Jon Blow and Brian Dennis. *"An Introduction to the Rush Language"* Proc. Tcl'94 Workshop. New Orleans, LA. June, 1994.

[Sat93] M. Satyanarayanan, et. al. *"Lightweight Recoverable Virtual Memory"* Proc. 14th Symp. on Operating System Principles (SOSP'93).

[Sto90] Michael Stonebraker. *"On Rules, Procedures, Caching, and Views in Database Systems"* Proc. ACM SIGMOD'90 Conf. on Management of Data.

[WLG93] Robert Wahbe, Steven Lucco and Susan Graham. *"Practical Data Breakpoints: Design and Implementation"* Proc. ACM SIGPLAN'93 Symp. on Programming Language Design and Implementation. Albuquerque, NM. 1993.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan Graham. *"Efficient Software-Based Fault Isolation"* Proc. ACM SIGOPS'93 Symp. on Operating System Principles. Asheville, NC. 1993.

[Whi94] J.E. White. *"Telescript Technology: the foundation of the electronic marketplace"* White Paper. General Magic, Inc. 1994.