A Replay Mechanism for Mostly Functional Parallel Programs

Robert H. Halstead, Jr.[1] David A. Kranz[2]

Digital Equipment Corporation
Cambridge Research Lab

CRL 90/6 November 13, 1990

Abstract

Performance measurement and debugging of parallel programs can both be aided by a *replay* mechanism that can reproduce prior, *traced* program executions by reading log information collected during tracing. The primary design goals of a replay mechanism should be to (1) produce correct results for any program and (2) keep the overhead of tracing small, so it does not seriously distort program execution. This is an extremely challenging pair of goals that no extant system meets completely.

This paper introduces a novel tracing and replay scheme—the *side-effect touch protocol*—that always meets goal (1), and also meets goal (2) for programs that are *mostly functional* (have few side effects). This scheme has been implemented in MulTVision, a system for visualizing the execution of parallel Multilisp programs. MulTVision imposes no tracing overhead on reads from memory (even from shared variables) and adds only a small amount of tracing overhead to Multilisp's touch operation. Greater overhead is added to side-effect operations, since they are presumed to be rare.

Measurements on MulTVision show that the overhead added for tracing is generally less than the overhead already incurred for task management and touch operations. While currently limited to program executions lasting seconds or less and creating at most tens of thousands of tasks, MulTVision is already useful for an interesting class of programs.

©Digital Equipment Corporation 1990. All rights reserved.

[1]DEC Cambridge Research Lab
[2]M.I.T. Laboratory for Computer Science

1 Introduction

Performance measurement and debugging of parallel programs are both challenging. Aggregate performance statistics such as total run time give little information about the efficiency of individual parts of a program or the location of bottlenecks. Debugging is often complicated by the presence of (possibly unintended) nondeterminism, making it difficult to reproduce an erroneous execution for investigation.

Both performance measurement and debugging can be aided by a visualization tool that helps a programmer understand what happens during a specific execution of a program. A programmer using such a tool first runs a program in *trace mode*. Trace-mode execution yields whatever output the program produces when executed in *production* (untraced) mode, but also yields as a by-product a *trace file*, or *log*, which records the history of the traced computation. A visualization interface (the *debug monitor*) offers a way to browse through the log and extract information about the traced execution.

MulTVision is a visualization tool for programs written in Multilisp [9, 10, 11, 12], which is the Lisp dialect Scheme [1, 23] extended with the `future` construct [3] for concurrency. Multilisp's computational model is essentially that of a shared-memory multiprocessor in which tasks interact through shared objects.

Determining what information to log is a difficult design decision in systems for visualizing parallel executions. Minimizing the trace-mode cost of generating the log is important to strengthen the correspondence between the timing details of trace-mode and production-mode execution, enhancing the system's usefulness for performance visualization. It also reduces the risk that tracing will change the program's qualitative behavior in a significant way (e.g., cause a pesky bug to disappear). On the other hand, the purpose of generating a log is to "freeze" a particular program execution and preserve it for detailed study: enough information must be logged to answer a wide range of questions from the user. An alternative to explicitly logging all information of interest is to log only enough information to characterize the nondeterministic choices made during the traced execution. A *replay engine* can then be guided by the log to follow the same path as the traced execution and regenerate the states through which the traced execution passed. Typically, a *partial replay* is performed to regenerate a state of interest, which can then be examined interactively. P-sequences [4], Instant Replay [18, 19], and PPD [20] all use this approach.

However, the opportunities for nondeterministic choices are so numerous in shared-memory parallel processing that developing a cheap tracing strategy allowing faithful replay of arbitrary programs has been difficult, and existing systems make compromises. PPD's logging strategy depends on compile-time identification of variables shared between tasks. Constructs, such as pointers or arrays, that preclude identification of a small number of shared variables will cause PPD's logs to become hopelessly enormous. PPD's strategy is thus unattractive for Multilisp programs, which use pointers heavily.

P-sequences and Instant Replay, by contrast, instrument synchronization objects

(semaphores in the case of P-sequences, arbitrary linearizable objects in the case of Instant Replay) to record necessary log information. These systems assume that all nondeterminism in programs comes from the order in which operations are performed on these objects. If a program, perhaps inadvertently, accesses shared variables outside the control of such synchronization objects, replay may be inaccurate.

Most existing Multilisp applications are mostly (or wholly) functional: they have few *true side effects* (writes to previously existing data structures that are—or may be—shared between tasks).[1] These applications are mostly functional not just because Multilisp and `future` support this style well, but because it is far easier to write a correct, highly parallel program this way.

MulTVision capitalizes on the mostly functional nature of typical Multilisp applications to make a different performance tradeoff than other systems. It adds a significant overhead burden to performing side effects[2] and to using values they store (both rare operations in mostly functional programs), but in return, MulTVision guarantees correct replay for *all* Multilisp programs, not just those written using a certain style.

The essence of MulTVision's approach is that side-effect operations in trace mode are modified to (1) log a *side-effect event* and (2) store a distinctive *side-effect value* encapsulating the actual value stored by the side-effect operation. No overhead is added to trace-mode memory read operations, but every touch operation[3] executed in trace mode checks to see if its operand is a side-effect value and, if so, logs a *side-effect touch event* and returns the value encapsulated within the side-effect value. (This check adds little cost because every touch operation already has to check its operand's type code, to see if it is a future.) The resulting log contains enough information to provide a completely faithful replay. Since only side effects and touches of the values they store are subjected to trace-mode overhead, tracing of mostly functional programs is efficient; in particular, *no* overhead is added to ordinary memory reads in trace mode, even if the location read is a shared variable on which side effects could be performed. (It is interesting that starting from a mostly functional base may enable construction of a debugging environment that provides services that could never be provided efficiently on a C or FORTRAN base that emphasizes side effects.)

1.1 Organization of MulTVision

The visualization component of MulTVision has two parts: the *debug monitor* and the *replay engine*. Both parts read the log produced by a trace-mode execution of a program, which contains events recording every instance when a task was created,

[1] Allocation and initialization of a new data object, although it involves writing to memory, does not involve true side effects, since the data object does not become accessible to other tasks until its initialization is complete, and hence no nondeterminism can result from these writes.

[2] In this paper, we informally use "side effect" to mean "true side effect."

[3] See Section 2.

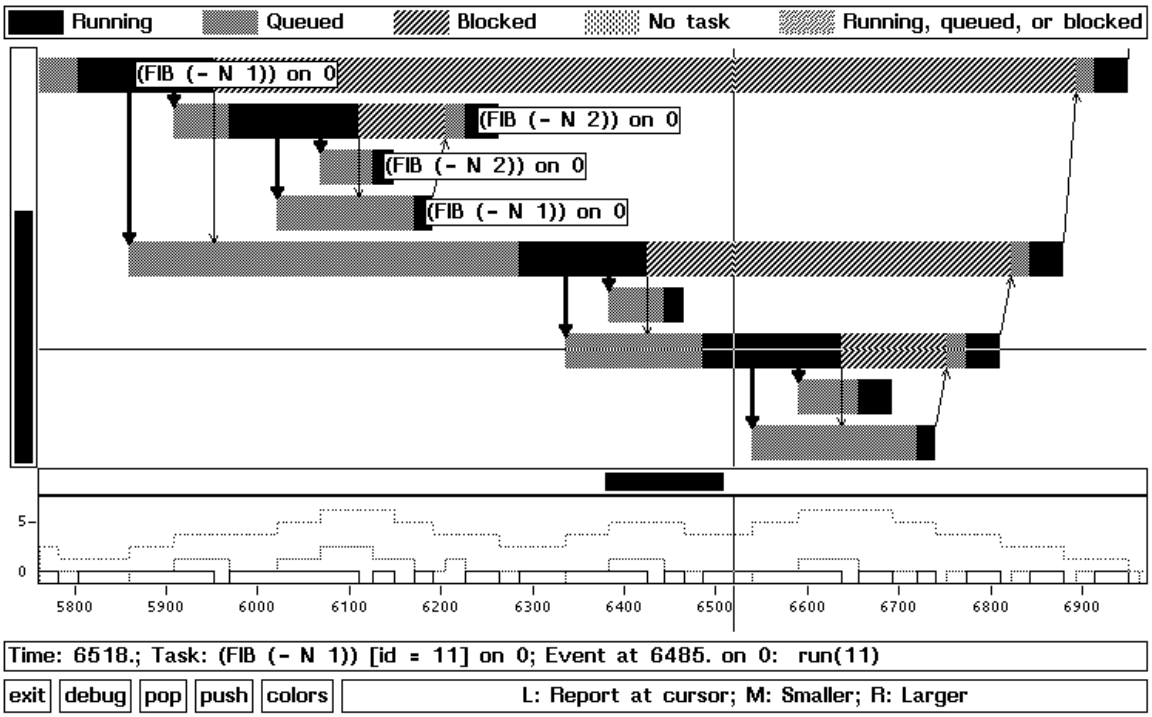

Figure 1: The MULTVISION debug monitor's display.

terminated, began to execute on a processor, or became blocked, in addition to the side-effect and side-effect touch events discussed above. The debug monitor graphically displays the execution of the traced computation and allows the programmer to examine parts of that execution. (See Figure 1.) Every logged event contains a *timestamp* derived by reading a real-time microsecond timer, used by the debug monitor to display events in their true temporal sequence.

The replay engine is built into the Mul-T system [14] on which MULTVISION is built and allows any data value generated during the traced execution to be regenerated for inspection. Such inspection is obviously useful when debugging, but can also help in performance visualization by identifying the inputs of specific computations shown in the debug monitor's display.

1.2 Overview of the Paper

Section 2 introduces relevant Multilisp concepts and Sections 3 and 4 explain our definition of "replay." Sections 5–9 explain MULTVISION's log protocol and its implementation. Section 10 gives preliminary performance figures, Section 11 reviews related work, and Sections 12 and 13 offer some perspective on the material presented in this paper.

2 Multilisp, Futures, and Mul-T

By default, execution of Multilisp programs is sequential (and follows the semantic rules of Scheme [1, 23]). Concurrency is introduced using the `future` construct: if X is a Multilisp expression, (`future` X) is an expression that evaluates X in a separate *task*. Without waiting for that evaluation of X to terminate, the expression (`future` X) returns a *future*—a placeholder that stands for the value that X's evaluation will eventually yield. When evaluation of X yields a value, the future *resolves* to that value, essentially losing its identity as a future and becoming the value of X.

Non-strict operations such as passing a parameter to a procedure, returning a result from a procedure, initializing a data-structure element, or storing into a memory location can use an unresolved future as an operand. However, *strict* operations such as addition and comparison (or any operation that requires information about its operand), if applied to an unresolved future, must block and wait for the future to resolve. (If applied to a resolved future, strict operations simply use the value to which the future resolved.) Hence, a strict operation must check each of its operands to see if it is an unresolved future, and if so the operation must block pending the resolution of the future. This check and possible suspension of execution is known as *touching* the operand. Note that resolving a future does not count as a side effect: a future can only resolve once, and any task requiring the future's value will block until the value is ready, so no nondeterminism can result.

Mul-T [14] is an implementation of Multilisp that was built by extending the Yale T system [22] with an implementation of the `future` construct. In Mul-T, futures are distinguished from all other types of data by giving them a special type tag. All strict built-in operations include code to touch their operands. Mul-T's optimizing compiler Orbit [15, 16] generates in-line code for many simple built-in operations. The type checks for the associated touch operations are also generated in-line, but if a touched operand is found to be a future, then control is transferred to an out-of-line subroutine to check further whether the future is resolved, and either suspend the current task or return the value to which the future resolved, as appropriate [14].

3 Replay

What does it mean to replay an execution of a program? Intuitively, it means to re-execute every operation performed during the original execution, with the same operands. The meaning of "same" here is subtle, however, since objects may have been created and then discarded during the original execution. Such objects must be "re-created" during the replay. These re-created objects should be indistinguishable, for all practical purposes, from the corresponding objects created during the original execution, but they are not identically the same objects.

It is only important to replay the programmer-visible aspects of a computation. In other words, if a traced computation built and manipulated a collection of inter-referencing objects, the replay must build and manipulate a corresponding collection

of objects, isomorphic to the original, and manipulate them in a corresponding way. But the replay need *not* reproduce the exact structure of task queues and other objects hidden within the implementation, except as required to perform corresponding operations at the programmer-visible level.

A satisfactory formal definition of the correspondence we desire between a traced execution and its replay would not fit in this paper, so we define the correspondence informally here. A correct replay E' of a traced execution E must *correspond to* E by containing a unique operation that is the image of, or *corresponds to*, every (programmer-visible) operation in E. A similar correspondence must exist between the data objects manipulated by E and E'. Corresponding objects must be observationally equivalent (e.g., if one is a number, the other must be the same number) even though, as discussed above, they will not always be the same object. Finally, if a replayed operation I' in E' corresponds to the operation I in E, then the objects on which I' operates must correspond in the obvious way to the operands of I. In this paper, we say "replaying the operation I" to mean, "during replay, performing the operation I' that corresponds to the operation I performed during the traced execution."

4 Nondeterminism

No special mechanism is required to replay a deterministic program; it suffices just to execute the program again, stopping at the point where inspection of run-time data values is desired. Thus, a log is needed only for replay of nondeterministic programs. But what does "deterministic" mean in this context? Informally, "deterministic" means that a program always performs the same computation: the meaning of "deterministic" thus turns on what we look at when deciding whether two computations are the "same."

We define "same" in terms of those aspects of the computation classified in Section 3 as "programmer-visible." Values of variables and data-structure slots, for example, are considered visible; the numerical value of the free-storage allocation pointer, the contents of physical processor task queues, and timing details not forced by precedence constraints among operations are examples of items that we choose to consider as invisible. We use the term *visible nondeterminism* to refer to nondeterminism in a visible aspect of a computation. A computation without visible nondeterminism might still have "invisible" nondeterminism, for example, in the assignment of tasks to processors for execution, but such nondeterminism is of no consequence to us, since reproducing all visible nondeterministic choices will, by definition, reproduce the programmer-visible aspects of the computation state during replay.

We can now clarify what is "true" about "true side effects": they are writes to memory that have the potential for causing visible nondeterminism. The trace file must record enough information about a traced execution that all visible nondeterministic choices made during the traced execution are made in the same way during replay. Visible nondeterminism can have several sources:

- Write-write conflicts, wherein the value found in a memory location at some point during program execution represents a nondeterministic choice between the values written by several write operations to that location.

- Read-write conflicts, wherein the value obtained by reading a memory location represents a nondeterministic choice between several values written to that location during program execution and possibly a pre-existing initial value in that location.

- Input and output conflicts, where input or output operations can be interleaved nondeterministically. These conflicts can be viewed as read-write conflicts on pointers that point to the current positions in input or output streams.

  Some inputs result from real-time interaction, where the result of a program operation depends on the time at which it is executed, relative to events in the external world. Examples of this kind of operation include reading a real-time clock and inquiring as to the availability of input on an interactive input stream. MULTVISION does not currently support real-time input operations, but logging the results of such operations would be a natural extension of our approach.

The programs we have focused on while developing MULTVISION are primarily computational, rather than input/output-intensive. Neither input nor output conflicts are significant in these programs, so we have concentrated on write-write and read-write conflicts as the important sources of nondeterminism to log. However, since input and output operations are normally less frequent than side effects, and (usually) are already considerably more expensive than side effects, tracing these operations should not introduce excessive overhead, compared with the overhead of tracing side effects. They could be traced either by viewing them as side effects on a stream pointer, as suggested above, or by other means, such as recording explicitly the data returned by an input operation.

5 Log Protocols for MULTVISION

Our major assumptions about our domain of mostly functional programs are

- True side effects are rare.

- Reads from potentially side-effectable locations are frequent, but reads from locations that have actually had true side effects performed on them are rare. For example, any pair object can have a side effect performed on it using the `set-car!` or `set-cdr!` operations, but typically, few pairs are side-effected in this way. Thus, while `car` and `cdr` operations to read the contents of pairs are usually frequent, few of those operations actually read a value stored as the result of a true side effect.

From these assumptions, and our desire to minimize the overhead introduced while executing in trace mode, we conclude that it is tolerable to write an event to the log every time a true side effect is performed, but it is highly undesirable to have to log an event every time a read is performed. On the other hand, we must log events recording the result of reading locations that have been side-effected, because these read operations may participate in read-write conflicts.

How can we tell whether a side effect has been performed on a location that is being read? We answer this question by giving an example log protocol, the *side-effect read protocol*, that has some of the properties we desire but is not actually the protocol used by MULTVISION. Then we explain the more intricate *side-effect touch protocol*, which is the protocol actually used.

6 The Side-Effect Read Protocol

Consider a side-effect operation S that writes a value V into a location L. In trace mode, S writes into L a *side-effect value* that contains V and also a unique *version ID* identifying which *version* of L this value is. S also logs a *side-effect event* recording this version ID. Every read operation performed in trace mode checks whether the value read is a side-effect value. If so, a *side-effect read event* is logged, recording the version ID found in this side-effect value.

Every time we replay a side-effect operation S, we must locate the side-effect event that was logged by S. Suppose S wrote a value V into a location L and logged a side-effect event carrying the version ID U. If, during replay, the corresponding side-effect operation S' writes a value V' into a location L', we can infer that V' corresponds to V (and L' also corresponds to L). Furthermore, suppose a logged side-effect read event with the same version ID U was produced by a read operation R that read V from L; the corresponding replayed read operation R' should be a read of L' that yields V'.

Interestingly, it is not necessary to write the actual value V to the log when logging the side-effect event: as long as the replay is carried out faithfully, when the corresponding replayed side-effect operation is executed, its operand V' is guaranteed to correspond to V. Since the same version ID U appears in every event that references V, simply associating V' with U during replay will provide a means to find V' whenever it is needed, without any representation of the value V ever having been written to the log! This association can be recorded in a *version database*, to which information is added every time a side effect is replayed, and which is consulted every time a read operation that logged a side-effect read event is replayed.[4]

[4]Other replay algorithms for the side-effect read protocol are possible. For example, instead of using a version database, reads and writes to a given location could be forced to occur in timestamp order, using a strategy similar to that of Instant Replay [18, 19]. This approach would be simpler and use less space, possibly at the cost of increased execution time. But since MULTVISION does not actually use the side-effect read protocol, these issues are moot: replay using a version database

The fact that values do not have to be included in logged events has two useful consequences. First, there is a time savings, since Multilisp programs often work with complex, structured values that would be expensive to dump. Second, since the log does not contain actual memory addresses in any form, replay need not use the same heap size or memory configuration as the traced execution.

Thus far, we have glossed over the question of how we decide which logged event (if any) corresponds to a given operation executed during replay. If every instruction in a program were executed at most once, establishing this correspondence would be easy: just include the instruction's address in the logged event, and when replaying an instruction, search for the logged event with the matching address.[5] However, real programs contain structures such as loops and procedure calls, so an instruction may be executed many times with (possibly) different operands. We need to match up the corresponding *instances* of executing each instruction. This can be done by maintaining an *instruction counter* and including its current value in each logged event that is used by the replay engine. The instruction counter need not be incremented for every instruction executed; it need only be incremented frequently enough so that no two events logged by the same task can have the same instruction count.

Tasks may execute concurrently and asynchronously. It thus makes sense for each task to have its own instruction counter, initialized at 0 when the task is created. Each logged event then includes a task ID as well as an instruction count, and together they uniquely identify the instruction instance that logged the event. For easy lookup in the log, however, tasks during replay must carry the same ID's as their corresponding tasks did during the traced execution. This means that *task creation events* must also be logged. Each such event indicates the ID given to a newly created task, and every task creation operation during replay must find the matching task creation event in the log and take the new task's ID from that event.

It is worth noting that if every instruction deterministically either generates or does not generate a log entry, then an instruction counter is not needed: each replayed instruction that should match an event can simply take the next logged event. Thus, the fact that a replayed read operation cannot tell (except by consulting the log) whether or not there should be a matching logged event is what makes the instruction counter necessary.

7 The Side-Effect Touch Protocol

The side-effect read protocol has the desired property that only true side effects and references to the values that they wrote create log entries. (Task creation also generates log entries, but the MULTVISION debug monitor needs task creation events

is introduced here to pave the way for explaining the side-effect touch protocol that MULTVISION actually uses.

[5]This strategy would require that the instruction's address not change between the traced execution and the replay—not necessarily true in MULTVISION.

anyhow, to generate its display.) Thus, a functional program, completely free of side effects, will log no extra events for replay. However, the potential logging of side-effect read events still burdens read operations during trace mode with checking for side-effect values, even if no side-effect read events are actually logged. An alternative, used in MULTVISION, is to defer checking for side-effect values until touch operations. Read operations in trace mode then work exactly as in production mode, but every touch operation has an extra duty: if it finds a side-effect value W as its operand, it logs a *side-effect touch event* recording the version ID found in W, and then returns the data value encapsulated within W. Touch operations that encounter futures still perform the same operations as in production mode.

The informal justification of the side-effect touch protocol is as follows. Visible nondeterminism can arise out of a read operation only when there is a read-write or write-write conflict and one of several values might be returned by the read. However, the only way that a nondeterministically chosen value can cause nondeterminism in the execution of a program is if the program contains an operation that acts differently according to what the value is. Any such operation is strict and hence touches its operands before using them. Therefore, knowing the nondeterministic outcome of the read operation at the time the value is touched is as good (for replay purposes) as knowing it when the value is read.

During tracing, it is possible to have a future that resolves to a side-effect value, a side-effect value that contains a future, or, in fact, an arbitrary chain of futures and side-effect values. A touch operation on such a chain must follow the chain until it finds a value that is neither a future nor a side-effect value. Since such a chain may have more than one side-effect value in it, a single touch operation may generate more than one side-effect touch event. During replay, we need to match logged side-effect touch events correctly with the steps along the touching chain to which they correspond. Therefore, during tracing the instruction counter is incremented once before every touch operation and once for each step down the touched chain: for every time that a touch finds that its operand is a future or a side-effect value and recursively begins another touch, the instruction count is incremented again.

Checking for side-effect values during touch operations rather than read operations is desirable because of the way Mul-T implements side-effect values and touch operations. Side-effect values in Mul-T carry the same basic type tag as futures, but are distinguished from futures by a subsidiary type tag. Thus, when the touched operand is neither a future nor a side-effect value, touch operations in trace mode are no more expensive than in production mode (except for incrementing the instruction counter). When the touched operand is a future, there is the extra cost of distinguishing it from a side-effect value before dispatching to the appropriate handler, but touches of futures happen far less often than reads from memory, making our approach cheaper than generating side-effect read events. Moreover, by the time a touch operation sees that it is handling a future (or side-effect value), control has already been transferred to out-of-line code, while any checking during read operations would either have to be coded in-line (greatly expanding code size) or introduce an out-of-line call for every read operation (seriously reducing performance in trace mode).

7.1 Read Values

During replay using the side-effect touch protocol, a read operation needs to return a value that, when touched, will trigger a check for a matching side-effect touch event in the log. We handle this requirement by using *read values*, which (like side-effect values) are "future-like." A touch operation on a read value traps to a special handler that performs the appropriate check in the log. If no matching event is found, then the initial value of the read location (at the beginning of the traced execution or when the object containing the location is initially created) should be returned. Consequently, every read value holds this initial value encapsulated within it.

When a location is read, there is no simple way to tell whether the location is ever side-effected. Therefore, every read operation during replay returns a read value, and every readable location conceptually contains the read value that will be returned when that location is read. MULTVISION's way to create the appearance that every readable location contains a read value is to lazily coerce the contents of any location that is read, to make it contain a read value if it doesn't already. A read operation then works as follows: if the read location contains a read value, just return it; otherwise, construct a read value containing the read location's current value, store that read value in the location, and also return it. During replay, we can create new objects with read values already in the objects' slots (on the assumption that most slots will eventually be read) but use lazy coercion to create the needed read values for pre-existing locations.

7.2 The Replay Algorithm

A MULTVISION replay begins by creating a version database from the events recorded in the log. Events are first classified by task ID and a collection of *version-database entries*, each corresponding to an event logged by that task, is created for each task ID. Each version-database entry includes the event's instruction count, timestamp, and a pointer to a *version object* that is shared by all version-database entries whose event had the same version ID. Each version object contains

- a *value* field to hold the actual value associated with this version;
- a *location* field to hold the read value found in the location where this version is written; and
- a queue of touchers waiting for this version's value.

These fields are all initially empty.

Every time a touch operation in replay mode finds a read value as an operand, it looks for a version-database entry that matches its task ID and instruction count. If such an entry is found, the touch operation returns the value field of the associated version object; if the value field is empty, the touch operation is suspended

and enqueued, awaiting this version's value. If no matching version-database entry is found, the touch operation simply returns the initial value encapsulated in the read value. (When a read value is touched and no matching log entry is found, the instruction count is *not* incremented, since this read-value touch does not correspond to a side-effect-value touch during the traced execution.)

Every replay-mode side-effect operation locates the version-database entry that matches its task ID and instruction count. The value to be written by that side effect is then stored into the associated version object (along with the read value found in the side-effected location) and any touchers awaiting this value are awakened. Interestingly, the side-effected location is not written when operating in replay mode, since any touches of side-effected versions of the location's value are recorded explicitly in the log! Also interesting is the fact that side-effect operations during replay are not required to occur in the same order as during the traced execution. (This reordering of side effects is permissible because each use of a value stored by a side effect indicates explicitly which version is used.) In fact, the ordering constraints on replay are so loose that it need not even use the same number of processors as were used in tracing. This fact can be exploited either to speed replay by using more processors or to reduce replay's impact on other system users by using fewer processors.

Side-effect events in the log do not identify the side-effected locations, yet accurate replay is still possible because the task ID and instruction count that are recorded establish the correspondence between the side-effect operations performed in trace mode and those performed in replay mode. However, it is convenient, while debugging, to be able to inquire about the current value of a location after a partial replay. For this purpose, it is useful to be able to identify the side-effect event, for a given location, having the largest timestamp less than the time of interest. Since a side-effect event in the log does not identify the side-effected location, the identity of that location must be recorded when the side-effect operation is replayed. This is done by having each side-effect operation fill the location field of its associated version with a pointer to the read value (lazily created if necessary) found in the side-effected location. Then the side-effect events pertaining to a location L can be found by searching the version database for entries that point to a version whose location field matches the read value found in L. When read values are used in this way, it is important to create only one read value for each location, so that the read values in two version objects are equal if and only if they are versions of the same location. (This imposes some synchronization constraints on the operation of lazily coercing a location's contents to a read value during replay.)

8 Partial Replay

MulTVision has two procedures for performing a partial replay. (`replay-to-time` *time*) replays a prefix of the traced execution containing all events whose timestamp field is less than *time* and no events whose timestamp field is greater or equal. Thus, every task that finished executing before *time* will be replayed in its entirety; every

other task will be replayed up to the instruction that matches the first event logged by that task with a timestamp greater than *time*. Conceptually, every replay begins at the initial state of the computation; however, if a replay has just been performed to a time T and a replay to a time $T' > T$ is then desired, `(continue-replay` T'`)` will start with the already available state at time T and only replay the portion of the computation from T to T'.

The intent of `replay-to-time` is to reproduce the situation as it existed at the time specified. Creating that situation exactly will not be possible, since some tasks will have no event at *time* and such tasks will have progressed all the way to their next logged event, but `replay-to-time` does ensure that all side effects occurring before *time*, and only those side effects, will be replayed.

`replay-to-time` is useful for determining the state of mutable data structures at a particular time. It can also be used to examine the state of any task at the point where an event was logged: just replay up to that event's timestamp and then use the Mul-T debugger to look around. However, it is sometimes useful to step through a task's history in more detail, so we also plan to provide `(replay-to-instruction` *taskID icount*`)`, which allows a particular task to be replayed to the precisely defined point where its instruction count was incremented to *icount*, allowing inspection of the task's local, immutable variables (e.g., stack frames) that might be deallocated before the next event logged by the task.

One approach to implementing `(replay-to-instruction` *taskID icount*`)` first estimates a time T by interpolating between the timestamps of the two events in *taskID* that bracket *icount*. Then, a minor variant of `(replay-to-time` T`)` is performed, in which all tasks other than *taskID* are replayed to time T, but *taskID* itself is replayed only up to its last event preceding T. Then *taskID* is stepped carefully until its instruction count reaches *icount*. (This will require some modification of the replay-mode code sequence for incrementing the instruction count.)

When examining a computation state generated by a partial replay, determining the correct value to attribute to read values is a major problem. If a read value R is found in a heap memory location, the version database must be searched for versions that have R in their location field. The value to attribute to R is the value found in the version created by the side-effect event with the most recent timestamp. A read value found elsewhere—for example, in a register, stack frame, or even in the value field of a version-database version—presents more of a problem. Such a read value represents a value that has been read but not yet touched. If the location in which the read value originated is in fact subject to side effects, we cannot tell which version of the location's value this read value represents until the read value is touched and we can see the associated side-effect touch event.

The only thing we can say with certainty about such a read value is that whatever value it actually represents has not yet had a chance to influence the flow of the computation (since any decision based on examining this value would have entailed a touch operation). This is useful information, but not being able to see the value may still prove frustrating.

If after a replay to time T the version database contains no versions referencing a given read value R, then it is safe to assume that the location from which R came has not been side-effected before time T. Consequently, it is safe to assume that R stands for its own initial value anywhere it is found. While this observation will help in some situations, a more aggressive solution would be to implement a "protocol converter" replay mode that reads a side-effect touch log and yields a side-effect read log, which could then be replayed without the problems of interpreting read values. We have been waiting until we can assess the magnitude of these problems in a practical setting before exploring this option, but it is clear that such a protocol converter could be implemented.

9 Structure of Compiled Code

The Mul-T compiler was modified to produce three different versions of object code for production-, trace-, and replay-mode execution. These versions differ, as described above, in the code sequences generated for a side effect, a touch, or a read from a potentially side-effectable location. The task creation code also has three different versions—in trace mode, the task ID of the created task is written to the log; during replay, this task ID is looked up and used as the task ID of the created task. Where an operation (such as a touch) has little to do in some cases but much to do in others, the generated code checks for the simple case and handles it in-line, but calls an out-of-line subroutine if a complex case is detected. Operations such as trace-mode or replay-mode side effects, whose handling is always complex, are coded simply as a call to an out-of-line subroutine.

Precompiled library procedures such as `length` and `print` present an interesting code management problem. Most such procedures perform touches, memory reads, or even side effects and thus need to be compiled differently for trace- and replay-mode execution. Whenever a program calls a library procedure, it must call the version compiled for the program's current execution mode.

We solve this problem using the *locale* feature [22] of the T system on which Mul-T is built. We create three locales, one for each execution mode. Each compiled version of a program is loaded into the locale corresponding to its execution mode. The library procedures compiled for each execution mode are bound to their names in the corresponding locale and thus are called only when that execution mode is active.

10 Performance

Since the initial implementation of MulTVision's replay engine has only recently become functional, performance figures presented here are necessarily preliminary. The value of MulTVision as a debugging and tuning tool will not be clear until it has been used on some real Multilisp application programs, and correctable deficiencies in

Expression	Prod. exec.	Trace exec.	Trace dump	Replay load	Replay exec.	Log bytes
1. `(touch (future 0))`	90	160	540	3000	420	78
2. `(set-car! x 0)`	0.13	16	130	770	50	18
3. `(touch (car x))`, side effect	0.87	16	110	760	38	18
4. `(touch (car x))`, no side effect	0.87	1.2	0	0	8	0

Table 1: Preliminary MULTVISION performance measurements (times in microseconds)

MULTVISION exposed by that experience have been corrected. Nevertheless, some basic data about MULTVISION's performance when executing primitive Multilisp operations are presented in Table 1. For each of four simple expressions, this table gives the execution time in production mode ("Prod. exec.") and trace mode ("Trace exec."); the time required, after a trace-mode execution, to dump the resulting log entries to disk ("Trace dump"); the time required to read the log entries and add them to the version database prior to a replay ("Replay load"); the execution time in replay mode ("Replay exec."); and finally the number of log bytes generated by the expression ("Log bytes"). All times were measured on our Encore Multimax[6] multiprocessor, with 30 MHz NS32532 processors. (All times are averages over several executions, but the "Trace dump" and "Replay load" times, which involve file-system accesses, were quite variable—no significance should be attached to small differences in these columns.)

The first expression in Table 1 is `(touch (future 0))`, which serves as a basic benchmark of task management performance: it requires one task creation, one touch operation, one task suspension, one resolution of a future, and one reawakening of a suspended task. The 78 bytes of log data for this expression include one task creation event and several other events used by the debug monitor but not by the replay engine. The expression `(set-car! x 0)`, where `x` is bound to a pair, is a benchmark of side-effect performance. The cost of side-effect touches is gauged by evaluating the expression `(touch (car x))` for the case where the `car` of the pair bound to `x` has been side-effected and for the case where it has not.

We can see from line 1 of Table 1 that the tracing overhead for creating a future is less than a factor of 2. (It is lower still in the common case where the future resolves before being touched.) A touch operation that does not discover a side effect also suffers tracing overhead (for incrementing the instruction counter) of less than a factor of 2, so the total tracing overhead for a side-effect-free program will be less than a factor of 2, and even less to the extent that the program emphasizes operations other than task creation and touches. The tracing overhead for side effects is a factor of more than 100, so a program densely populated with side effects will slow down

[6]Multimax is a trademark of Encore Computer Corporation.

dramatically when traced, but a program that averages one side effect or less per future (a reasonable standard for what it means to be mostly functional) will incur a tracing overhead for side effects that is a small fraction of the overhead already present for task creation. The time required to dump the log to disk after a traced execution is considerably larger than the tracing overhead itself, but it does not distort the timing of the traced execution.

Fast replay is less important to us than fast tracing. While faster replay leads to better interactive response, the ordering constraints on replayed operations are fixed by the log and are guaranteed to be observed no matter what speed the replay runs at. Once replay is fast enough for adequate user interaction, there is no need to speed it further, even if it is significantly slower than production-mode execution. This is fortunate, since Table 1 shows replay to be significantly slower than tracing. We await further experience with MULTVISION to see whether the replay times listed in Table 1 are too slow and constitute the limiting factor on MULTVISION's usefulness, but we are confident that they can be speeded up with some effort. (See Section 12.)

11 Related Work

Both MULTVISION's trace-file format and its debug monitor evolved from the ParVis system [2]. ParVis did not have a replay engine (though the idea of incorporating one is mentioned in [2]), but other researchers have developed systems for replay of parallel programs. These other replay projects can be classified first by the type of interprocess communication they were designed for: either message-passing or shared-memory.

BugNet [5], Radar [17], and Schiffenbauer's debugger [24] are three systems for replay in message-passing environments. All work by logging interprocess messages and can replay a process by feeding it its logged input messages. Groups of processes can be replayed in a similar manner. In this approach, the log contains the actual data values communicated between processes, so its size is proportional to the actual volume of information communicated.

MULTVISION fits into the shared-memory category. Previous work on replay of shared-memory parallel programs includes P-sequences [4], Instant Replay [18, 19], and PPD [20]. PPD divides programs into *E-blocks* and writes *prelogs* and *postlogs* containing the values of selected variables when entering and exiting E-blocks. Compile-time analysis selects the variables to include in prelogs and postlogs and identifies shared variables that require more frequent logging. Thus PPD, like BugNet and Radar, records actual data values in a log. Since shared-memory architectures enable quite high rates of communication between processors, PPD's logs can quickly get quite large. On the other hand, the mass of information recorded in the prelogs and postlogs makes it possible to answer most questions without performing a full replay from the beginning of the log, an advantage for interactive use. Any use of pointers that expands the set of variables that may be accessed in an E-block will cause major problems for PPD's approach.

P-sequences and Instant Replay share MULTVISION's approach of logging ordering information rather than actual data values. For Instant Replay, a program is viewed as operating on a set of objects whose history can be viewed as a sequence of *versions*. Every version is created by a *write* operation on an object; *read* operations retrieve data that is version-specific, but do not change the object they are applied to. The Instant Replay log indicates the sequence of accesses to, and creation of, object versions. Replay is guided by this log so that each object's versions are created in the same order, and each read operation reads the same version, as in the original execution. The implementor of the basic object operations (who may also be the application programmer) is responsible for ensuring that they are implemented correctly (they must be linearizable) and that they handle the log entries properly; otherwise, replay may be performed incorrectly. P-sequences are essentially a specialization of this approach to the case where all synchronization is performed using semaphores.

By logging only ordering information, rather than actual data values, P-sequences and Instant Replay achieve significant reductions in log size and concomitant reductions in the cost of writing and then reading the log. The price of this optimization is that answering questions about a computation's intermediate data values requires replaying the computation from the beginning. Recap [21] is a system that takes periodic checkpoints and also records a log: a replay to reach a given state can start from the most recent checkpoint instead of from the beginning, and hence can take less time. (Other systems that use checkpoints to optimize replay, but don't focus on parallel computing, include [25] and [26].)

MULTVISION's approach is closest to that of P-sequences and Instant Replay, but MULTVISION differs from them in that it does not require programs to adhere to special disciplines to ensure accurate replay.

12 Discussion

While the soundness of MULTVISION's replay algorithm is clear, the biggest unanswered questions concern the size of the program executions to which MULTVISION can be applied effectively. Four factors may limit this size:

- Replay time. Table 1 suggests that the execution time for replay will be about an order of magnitude greater than the time for production-mode execution, and the time for building the version database may be yet another order of magnitude greater. Since this is a simple, initial implementation, there should be considerable room for improvement in these figures, but with MULTVISION's current strategy of rebuilding the version database before every replay, replay of an execution that took even one second may be uncomfortably slow.

- Version-database size. Currently, the entire version database is built before replay begins, consuming storage proportional to the total number of task-creation, side-effect, and side-effect touch events in the log. This storage remains

allocated for the entire duration of the replay; if it exceeds a dozen megabytes or so in size, it will begin to seriously strain physical memory resources. Since each event accounts for tens of microseconds of replay execution time and hundreds of microseconds of replay load time, version-database size constrains are probably no tighter than the replay-time constraints discussed above; however, if replay time is improved through optimizations, version-database size could prove to be a bottleneck, motivating exploration of strategies for incrementally creating the version database as replay progresses and already-used portions of the version database can be deallocated and recycled.

- Trace-file size. A mostly functional 16-processor execution that takes one second in production mode and has moderate task granularity might create 40,000 tasks and generate on the order of 5 megabytes of trace data. Currently, this looks like a weaker constraint than replay time or version-database size, but it clearly imposes a limit—finding a file system with room for a 100-megabyte trace file is often difficult!

- Debug-monitor performance. While this paper is not about the debug monitor, it is currently far more expensive for the debug monitor to load a trace file for display than it is for the replay engine to build its version database from the same trace file. There is little point in worrying about the replay engine's performance on a trace that is too expensive to use with the debug monitor; thus, the debug monitor will probably be the first target of optimization efforts.

While the limitations listed above are significant, MulTVision is already valuable, even at the current performance level. Not all computations of interest can be performed on the time scale of seconds to which the current MulTVision implementation is probably limited, but today's multiprocessors are fast enough to perform quite elaborate computations on that time scale—more than elaborate enough to benefit from the detailed investigation that MulTVision allows. Moreover, many computations that take much longer than that can be scaled down to smaller problem instances with analogous properties. Investigation of these smaller problem instances using MulTVision can yield useful insights about the behavior of the larger computations. From our past experiences with Multilisp, we have questions about several problems of this kind that MulTVision is already a powerful enough tool to answer.

To expand MulTVision's range further, we can take the following steps:

- Careful performance measurement and tuning of the MulTVision implementation.

- Preprocessing of the log to produce checkpoints or other useful information to accelerate replay, such as a side-effect read log or preformatted version database.

- Improving the debug monitor's record-keeping. If the debug monitor carefully records all interaction with the replay engine, the user may be able to answer a question about a previously visited state without performing another replay to

revisit it: the answer may be found in the record of previous interaction with that state.

It would also be desirable to expand the set of programs that we can classify as mostly functional and apply MULTVISION to. A significant gap in the current paradigm affects programs that use arrays. Typically, Mul-T programs allocate arrays in an uninitialized state. Array initialization is then performed using side effects, generating a large number of side-effect events even if the array is used in a completely deterministic way (e.g., no other task sees it until initialization is complete). One solution to this problem is to introduce explicitly functional language constructs, such as the *array comprehensions* of Haskell [13], to use when creating arrays in a functional way. A more ambitious alternative is to develop compile-time analysis techniques that can identify these kinds of side effects as not being true side effects that require special treatment during tracing or replay.

A final question concerns ensuring that a replay starts with the same initial conditions as a traced execution. If the initial conditions are different, the replay obviously risks being inaccurate. This problem is exacerbated by the interactive nature of Lisp programming environments: a C program, for example, always begins execution with its data memory in a given state, but Lisp programs execute in a "workspace" where variable bindings may have been created or changed interactively before a traced execution. Automatically reproducing such interactive inputs before replay looks infeasible in the general case, but simple tricks may help a lot in real life. For example, when a side effect is performed during tracing and the value overwritten is not a side-effect value, the value overwritten could be recorded; if a replay is then performed during the same Mul-T invocation, the recorded initial value can be used. However, if replays are performed days later, there is no obvious option other than warning the programmer about the dangers of using the wrong initial conditions and advising either the taking of careful notes or specification of all relevant initial state in the Multilisp program source file where it is unlikely to be forgotten.

13 Conclusion

Our basic objective was to develop a replay mechanism for Multilisp programs that would produce correct results for *any* program and would add minimal tracing overhead to mostly functional programs. Both the side-effect read and side-effect touch protocols satisfy the correctness requirement, provided only that the user establishes the proper initial state before attempting a replay. Notably, neither protocol requires programs to observe any stylistic restrictions or be amenable to compile-time analysis to enable accurate replay. The side-effect touch protocol also satisfies the performance requirement by adding no overhead to reads and little overhead to touches, the two most common operations in mostly functional programs. This protocol and a replay algorithm for it have both been implemented in the MULTVISION system and their performance for basic operations has been measured. This performance is

good enough to be immediately useful, although (as with all replay schemes) better performance would add significant further value.

Plans for further development of MULTVISION must strike a balance between two goals: (1) efficiently handling larger and more varied program executions and (2) improving the visualization tools that can be applied to the program executions that are already handled efficiently. Most of the ideas in Section 12 are aimed at the former goal, but now we must accumulate enough experience with MULTVISION and substantial Multilisp programs to see whether ease of visualization or speed of tracing and replay is the most important next area for improvement.

14 Acknowledgments

We are grateful to Patrick Sobalvarro for his help in building the debug monitor and to Rick Mohr for helpful comments on this manuscript. Support for this research project was provided by Digital Equipment Corporation and by DARPA under contract number N00014-87-K-0825.

References

[1] Abelson, H., and G.J. Sussman with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, 1985.

[2] Bagnall, L., *ParVis: A Program Visualization Tool for Multilisp*, S.M. thesis, MIT E.E.C.S. Dept., Cambridge, Mass., Feb. 1989.

[3] Baker, H., and C. Hewitt, "The Incremental Garbage Collection of Processes," M.I.T. Artificial Intelligence Laboratory Memo 454, Cambridge, Mass., Dec. 1977.

[4] Carver, R., and K.-C. Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables," *6th Int'l. Conf. on Distributed Computing Systems*, May 1986, pp. 428–433.

[5] Curtis, R., and L. Wittie, "BugNet: A Debugging System for Parallel Programming Environments," *3rd Int'l. Conf. on Distributed Computing Systems*, Oct. 1982, pp. 394–399.

[6] Gabriel, R.P., and J. McCarthy, "Queue-based Multi-processing Lisp," *1984 ACM Symp. on Lisp and Functional Programming*, Austin, Tex., Aug. 1984, pp. 25–44.

[7] Goldman, R., and R.P. Gabriel, "Preliminary Results with the Initial Implementation of Qlisp," *1988 ACM Symp. on Lisp and Functional Programming*, Snowbird, Utah, July 1988, pp. 143–152.

[8] Goldman, R., R. Gabriel, and C. Sexton, "Qlisp: An Interim Report," *Parallel Lisp: Languages and Systems (Lecture Notes in Computer Science 441)*, (Proc. US/Japan Workshop on Parallel Lisp), Springer-Verlag, Berlin, 1990, pp. 161–181.

[9] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. on Prog. Languages and Systems 7:4*, October 1985, pp. 501–538.

[10] Halstead, R., "Parallel Symbolic Computing," *IEEE Computer 19:8*, August 1986, pp. 35–43.

[11] Halstead, R., "An Assessment of Multilisp: Lessons from Experience," *Int'l. J. of Parallel Programming 15:6*, Dec. 1986, pp. 459–501.

[12] Halstead, R., "New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools," *Parallel Lisp: Languages and Systems (Lecture Notes in Computer Science 441)*, (Proc. US/Japan Workshop on Parallel Lisp), Springer-Verlag, Berlin, 1990, pp. 2–57.

[13] Hudak, P., P. Wadler, et al., *Report on the Programming Language Haskell*, version 1.0, April 1990.

[14] Kranz, D., R. Halstead, and E. Mohr, "Mul-T, A High-Performance Parallel Lisp", *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 81–90.

[15] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "Orbit: An Optimizing Compiler for Scheme," *Proc. SIGPLAN '86 Symp. on Compiler Construction*, June 1986, pp. 219–233.

[16] Kranz, D., *ORBIT: An Optimizing Compiler for Scheme*, Yale University Technical Report YALEU/DCS/RR-632, February 1988.

[17] LeBlanc, R., and A. Robbins, "Event-Driven Monitoring of Distributed Programs," *5th Int'l. Conf. on Distributed Computing Systems*, May 1985, pp. 515–522.

[18] LeBlanc, T., and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Computers C-36:4*, April 1987, pp. 471–482.

[19] Mellor-Crummey, J., *Debugging and Analysis of Large-Scale Parallel Programs*, Technical Report 312, University of Rochester Computer Science Dept., Sept. 1989.

[20] Miller, B., and J.-D. Choi, *ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, Atlanta, June 1988, pp. 135–144.

[21] Pan, D., and M. Linton, "Supporting Reverse Execution for Parallel Programs," *Workshop on Parallel and Distributed Debugging (SIGPLAN Notices 24:1, January 1989)*, May 1988, pp. 124–129.

[22] Rees, J., N. Adams, and J. Meehan, *The T Manual*, fifth edition (pre-beta draft), Yale University Computer Science Department, October 1988.

[23] Rees, J., and W. Clinger, eds., "Revised^3 Report on the Algorithmic Language Scheme," *ACM SIGPLAN Notices 21:12*, Dec. 1986, pp. 37–79.

[24] Schiffenbauer, R., *Interactive Debugging in a Distributed Computational Environment*, M.I.T. Laboratory for Computer Science Technical Report TR-264, Cambridge, Mass., Sept. 1981.

[25] Tolmach, A., and A. Appel, "Debugging Standard ML Without Reverse Engineering," *1990 ACM Symp. on Lisp and Functional Programming*, Nice, France, June 1990, pp. 1–12.

[26] Wilson, P., and T. Moher, "Demonic Memory for Process Histories," *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 330–343.