IGOR: A System for Program Debugging via Reversible Execution

Stuart I. Feldman

Channing B. Brown

Bellcore
Morristown, New Jersey

ABSTRACT

Typical debugging tools are insufficiently powerful to find the most difficult types of program misbehaviors. We have implemented a prototype of a new debugging system, IGOR, which provides a great deal more useful information and offers new abilities that are quite promising. The system runs fast enough to be quite useful while providing many features that are usually available only in an interpreted environment. We describe here some improved facilities (reverse execution, selective searching of execution history, substitution of data and executable parts of the programs) that are needed for serious debugging and are not found in traditional single-thread debugging tools. With a little help from the operating system, we provide these capabilities at reasonable cost without modifying the executable code and running fairly close to full speed. The prototype runs under the DUNE distributed operating system.

The current system only supports debugging of single-thread programs. The paper describes planned extensions to make use of extra processors to speed the system and for applying the technique to multi-thread and time-dependent executions.

1. INTRODUCTION

The usual debugging tools are insufficiently powerful for finding the most difficult types of program misbehaviors. We describe here some improved facilities (reverse execution, selective searching of execution history, substitution of data and executable parts of the programs) that are quite promising, and can be supported at reasonable cost, but are unusual in a compiled-language environment.

The typical debugger provides two facilities, post-mortem dumping and breakpoints. Most modern debuggers will provide a multi-window menu-driven graphical interface and nice representations of the data in memory after the crash, but the basic data are the same as found in an old-fashioned octal dump. Breakpoints permit modification of the execution flow, and most commonly provide selective checking of assertions and occasional output of trace information at a fixed point in the program, once it is reinitiated. A careful autopsy followed by insertion of breakpoints will often locate an error, but deep bugs often resist these elementary techniques. A subtle malfunction can cause complicated results which do not result in a noticeable problem till much later in the computation, at which time the original source of the problem is no longer obvious. With sufficient pain and diligence, the usual tools can lead to a fixed program, but for sufficiently devious bugs, it is more important to provide better information about the course of the computation than to make the interface somewhat more attractive.

As a simple example of the sort of bug that can be maddening to identify correctly, consider a program in which a basic tree manipulation operation is misimplemented so that in some case a cycle appears in the data structure:

```
tree *p, *q;
p = q;
   . . .
p ->right_child ->left_child = q;    /* whoops */
```

The declared type structure is not violated by these assignments, but the result is a cycle. The problem will not become apparent till a full tree-walk is applied to this data structure, which will result in an infinite recursion or some similar unexpected collapse. If data structures are built up piecemeal in a program, it is possible that the proximate cause of the crash (the tree walk) will occur billions of instructions after the actual cause (the error in storing *q* into the tree structure.) Faced with a stack overflow in a depth first search routine, or a related symptom, the programmer will examine the final state of the computation (the post mortem dump) and discover huge numbers of loops, though this will take a lot of work unless the debugger has a good way of picturing graphs. The programmer wants to ask

"When did this tree first get an illegal cycle?"

The tricky parts of this question are: "when", "*this* tree", "first get", and "cycle". The first and third questions require being able to look back in time. The second requires identifying particular data structures, whose locations in memory will change with time but for which there must be some identifying property (location of the root of the tree perhaps). The fourth requires introducing a complicated test condition that is unlikely to be present in a standard debugger.

1.1. Prototype Debugging System

In order to help solve these problems, we have constructed a prototype system named IGOR that enables us to look at varying epochs of the run of a program and to make changes in the program and its data. The system can automatically save snapshots of the program's execution. It also has procedures for modifying these images, then bringing them to life. Reverse execution is not a new concept; cf. COPE system [1]. Earlier systems required very expensive support (full interpretation, generation of code with inversion options, or special recompilation to threaded code). We use a mechanism that could be supported on almost any virtual memory machine and requires no changes to the executing code. (In effect, we provide an automatic incremental checkpoint facility, or an automated recovery block [2] approach). It may not always be possible to restart a program if the external state cannot be re-created (it is difficult to rewind a printer or call back data from a communications line), but the tools can succeed in a wide variety of cases. For changing the program, we implement a dynamic loading facility. To allow special case examination of data, we provide an interpreter that can be applied selectively to certain procedures while permitting others to run at full speed. Data can also be modified at various states of the computation.

1.2. Implementation Environment

The debugger prototype described here was implemented, mostly in C, on a Motorola-68000-based system running the DUNE [3] distributed operating system (which supports the semantics of UNIX System V). Certain special system calls make this implementation economical; they are easy to add to almost any system with virtual memory. Some of the code depends on the way the 68000 handles stacks and registers and on the way this UNIX system represents certain data, but the ideas can be translated to similar systems. The work is simplified by assuming that programs are written in C, which has a transparent procedure call mechanism and a rudimentary scoping mechanism (there is a single external scope for all global variables and functions). We take advantage of these simplifications in the prototype implementation, which requires modifications to the compiler, loader, and standard library, but another procedural language could have been used with a little extra effort.

UNIX is a registered trademark of AT&T

1.3. Aims and Limitations

A major goal of this work was to provide a practical way to attack large and complicated problems. When the debugger is turned off and no checkpoints are being made, there is almost no effect on the execution of a program. The few modifications made to the compiler add just a few bytes to the program. The program runs somewhat slower when the history capture mechanism is functioning, but we judge the slowdown to be acceptable for real use. (Performance is discussed in a later section.)

The technique described here ought to apply to a broad spectrum of programs, but it is not a panacea. A program with bad working set properties (e.g., one which touches memory in unpredictable ways) will generate high output traffic. In a program that makes many small changes to huge files, it may be difficult to restore the earlier state of the files at low cost. Irreversible input/output (mouse movements, printer output) may also prevent full use of these techniques. The approach is untried for debugging asynchronous parallel programs, though it could be applied well in a synchronous distributed environment. Despite these caveats, it *is* applicable to most "garden variety" programs and can be of help for finding bugs that have some of the unfortunate properties described above.

The system described here is an experiment, not a finished product. There are therefore many loose ends that could be finished but are not essential to the initial study. Performance is quite acceptable though the system has not been lovingly tuned. A number of important cases of input/output behavior and the easier cases of tracing program properties have been covered; more general cases are understood but not implemented. We have concentrated on providing functionality rather than on a snazzy interface. Even so, IGOR is usable and actually helped find some bugs in the course of its own development.

1.4. System Changes

In order to apply IGOR, one should use a modified compiler, library, and loader, and run on a system with a modified kernel. The changes however are quite small. The compiler needs to put out a little extra information about data allocation and to insert no-op instructions at the start of each procedure. The basic start-up routine that is automatically loaded into a program must be changed to permit turning the history mechanism on and to simplify restarting. The relatively small kernel modifications are described in detail later.

2. RESTARTING EXECUTION

One principal ability of IGOR is restarting the execution of a program. This might be attempted after the abnormal termination of a program, or it might involve backing up to a previous point in the execution sequence and restarting from there. This section explores both possibilities.

2.1. Reviving Dead Programs

When a program terminates abnormally (such as when it attempts an illegal memory reference or divides by zero), it typically produces a memory dump (commonly called a *core* image by the UNIX operating system) which contains the user accessible state of the program (registers, static variables, stack, heap). We have developed a set of programs to enable a user to restart a program from a core image. The restart programs work in conjunction with a routine that is bound into the user's program whenever the debugging version of the compiler is used. Unless changed in the prestart procedure, the run picks up at the C statement after the one that caused the dump. Of course, there are some situations that cause a dump from which it is not possible to restart without effort. For example, if the program has run out of stack space due to an infinite recursion, the stack must be unwound before execution can proceed.

2.2. Restarting Execution from an Interim State

We next extended the concept of reviving a dead program to allow the revival of a program from various interim points between the start of execution and its (normal or abnormal) termination. This requires saving the state of the execution at interim points, often referred to as *checkpointing*.

2.2.1. Checkpointing

We have implemented a scheme in which the execution state is written out periodically to a set of files during the execution. (If execution runs long enough or the memory access pattern is malign, these files could grow rapidly, but in favorable cases the checkpoint files grow slowly.) In order to save interim execution states efficiently, we rely on a concept similar to that used in virtual memory systems.

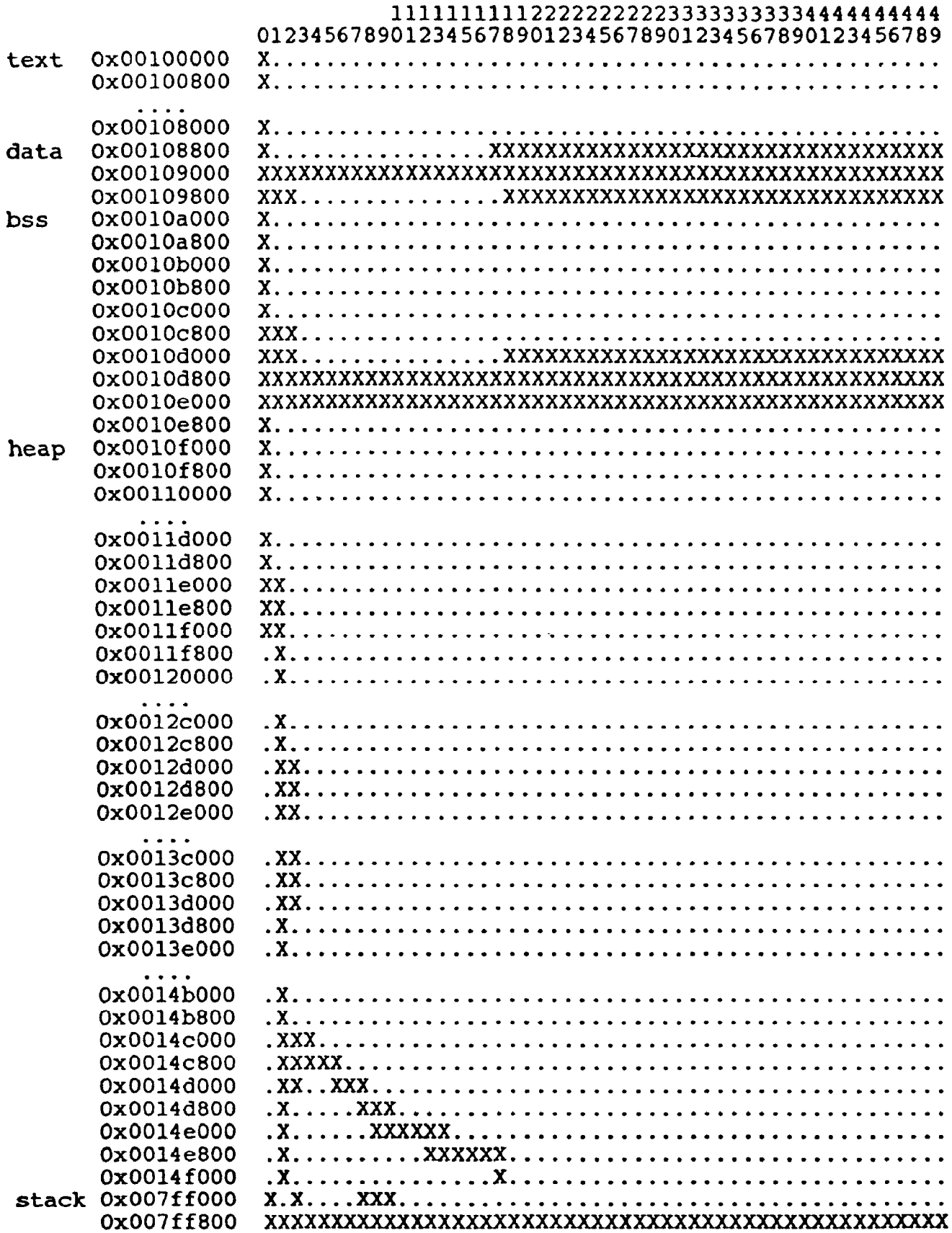
A virtual memory system generally divides the program's memory into fixed-size pages, keeps the more recently required pages in physical memory, and stores the remaining pages in secondary memory. When a new page needs to be brought into physical memory, the system may swap out an existing page to make room for it; if that page was not written on during its time in physical memory, there is no need to write it out to disk. Many processors assist the virtual memory system by marking each page that is written upon. (If the hardware does not maintain a "dirty bit" for each page, but does have page-oriented write protection, the software can get the same information by protecting pages against being written upon at the start of a time interval, then setting a bit and permitting future writes to the page when a protection violation is detected. Other uses of this technique are suggested later.)

At intervals, IGOR saves a copy of every page that has been changed since the last check. In effect, we simulate demand paging activity. If the program has good locality of reference, only a fraction of the pages will be changed during any short computation, while many different cells will be changed in a single page. We therefore expect that this technique will not waste too much time copying unnecessary information and will run fast by taking advantage of installed hardware.

The result of executing in this mode is to produce a file containing copies of all pages changed during each interval. It is then easy to reconstruct the memory state at any of those discrete times. To examine the state at an arbitrary time, it suffices to find the checkpoint immediately preceding that time, then execute forward (using an interpreter) till the specific program point is reached. If checkpoints are moderately frequent, then the number of instructions to be executed will be small and so the delay incurred by running a slow interpreter is in practice acceptable. There is a tradeoff between amount of page copying and timer interval, as is shown in the Performance section below. The interval is chosen in part based on how long the human user is willing to wait before the interpreter reaches the exact point of interest. If the interpreter runs around 100 times slower than real time, an impatient person who is only willing to wait a half minute would then set the interval to a quarter second.

Various indices and tables improve the use of the history data. To reconstruct a memory image at a particular checkpoint interval, it is necessary to choose the copy of each page that was changed during that interval or most recently before it. To locate the time when a particular static variable reached a particular value, it is only necessary to look at the generations of the page on which the variable resides.

An example shows the memory reference pattern we hope to see when using this technique. The following Figure shows the pages that were dumped at each checkpoint for an example execution of the UNIX *sort* program. The program was sorting a file of 2853 lines, and checkpoints were taken four times per second. The left side of the graph shows the hexadecimal address of the start of each page; the top of the graph numbers the checkpoints sequentially from zero. An X indicates that the page was dumped during the checkpoint. An ellipsis indicates a line that is identical to the preceding line. Note that a few pages (bottom of the stack and some external variables) are written each time interval, but that most pages change only occasionally after the first time. (Of course, some other programs do not behave this nicely, as will be seen in the section on performance.)



A Graph of Checkpointed Pages

2.2.2. Special System Calls for Checkpointing

In order to create a checkpoint at a given point during the execution, we use a new UNIX system call, *pagemod*, that returns a list of all memory pages that have been written since the previous call to *pagemod*. At each checkpoint, the contents of these memory pages are saved in the checkpoint files, with the contents of the registers and program counter. A timestamp (showing the program's elapsed real time, user time, and system time) is also added to indicate when the checkpoint occurred.

The interval between checkpoints is controlled with a call to another new system call, *ualarm*. This system call is similar to the standard UNIX *alarm* system call, in that it provides an interrupt to the program after a specified amount of time has elapsed. However, *ualarm* counts user CPU time rather than real time, and allows a finer granularity (0.01 second in place of 1.0 second for *alarm*). These times seem appropriate to our 68000-family processors, but might not be right for machines of enormously different speeds.

We provide a program that allows the user to modify the checkpointing interval or to turn the feature on or off by modifying the executable file.

2.2.3. Checkpointing Without the New System Calls

It would be considerably more costly, and in some ways difficult or impossible, to implement similar checkpointing without the two new system calls mentioned. If the *pagemod* call were not available but there were easy access to the protection map and traps, the system could write-protect all pages at the beginning of an interval, then copy and turn off write protection on each data page that suffered a write protection violation. Without virtual memory support, it would be necessary to write out all of memory at each checkpoint, or at least to touch all of the writable part of the user's memory at each checkpoint. (If necessary, it would be possible to save all of the data from the preceding checkpoint time, or a set of hash values corresponding to those pages; the data or a set of hashed values at the next interval could then be compared with those values.) If the program displays a reasonable locality of reference, then the number of pages written at any one checkpoint should be small compared to the program's total address space.

Without the *ualarm* call, the frequency of checkpointing might be inadequate for debugging purposes. In addition, an alarm interrupt based on real time elapsed would occur sometimes during a system call (as, for example, while the program is waiting for input from a terminal). Such an unexpected interrupt would cause the system call to fail, and the program would behave erroneously. An interrupt based on user time elapsed does not cause such a problem. It also ensures that the program has accomplished some work during each interval, avoiding superfluous interrupts.

Some results on the performance of IGOR are provided below.

2.2.4. Checkpoint data manipulation

After running a program with the checkpointing feature enabled, the user can search the checkpoint data for the values of a variable (or a memory address); IGOR responds with the values taken on by that variable during the course of execution, with the checkpoint number and timestamp for reference purposes.

Execution restart from a checkpoint is similar to restart from a core image. IGOR reconstructs a core image for the specified checkpoint by retrieving the latest image of each page as of the desired checkpoint. The data are then copied in from the reconstructed core image, the stack and registers are restored, and execution resumes.

3. DYNAMIC FUNCTION REPLACEMENT

IGOR also makes it possible to replace parts of the program dynamically during debugging. IGOR allows the user to replace one or more C functions with new versions without having to link the entire program, and to restart execution from a core (or checkpoint) image with the new versions of the functions. Fundamental problems arise if the frozen program is executing a procedure that is being changed or if data elements being replaced are referenced in an active procedure. We do the best we can to recognize these cases and avoid the side effects of changes to a suspended program.

The user can replace any function in a program with a new version; the new object module is incrementally loaded into the program by a special version of the loader. The old version of the function remains in the executable file, so the user can switch among several versions.

When the new version is loaded, the first few bytes of the old version are replaced with a jump to the start of the new version. To restore the old version, the original bytes are replaced. (To simplify the implementation, the debugging version of the compiler inserts enough no-op instructions to provide room for the jump to be inserted.) The same effect could be achieved on other systems by calling all procedures indirectly through a transfer vector.

If the function is replaced in a memory image that is to be used for an execution restart, it is necessary that the variables in the new version retain their values from the previous version. For static and external variables, the loader accomplishes this by using the address of the original variable for the new version, rather than allocating a new memory location for the variable. Of course, any new variables introduced by the new version are allocated space in the usual way.

If the user replaces a function that is active at the execution restart point, the new version does not apply to the currently active instantiation(s); i.e., the new code is not used until the next time the function is called. Thus, automatic variables need not be tied to a previous version.

The loader notes various differences between the declaration of a variable in the old and new versions of a function, and warns the user about such discrepancies. For example, the type of a variable or the size of an array might change; in general such a change will lead to erroneous results. Ideally, a system might attempt to resolve such contradictions, but such considerations can lead to the undecidable problem of program equivalence.

If a function is replaced in an executable file that is to be used for an execution restart, the restart and replacement must be carefully coordinated. The incremental linking cannot occur until after the data segment is copied in from the core image. IGOR then knows what parts of memory have been allocated during execution, so that the new function can be allocated to an unused portion of memory. However, the incremental linking must occur before the execution begins.

4. RESTARTING INPUT/OUTPUT

In order to restart the execution of a program from a previous state, it is necessary to restore the input/output environment to its previous state. In general this is difficult, and is impossible in some cases. (One cannot undo the writing of output data to a hardcopy device.)

We have implemented some techniques that allow the re-establishment of the input/output environment under various conditions, as described in this section. In each case, a prestart routine is run before the user's program execution resumes and it attempts to restore the status quo ante. Basically, a program which does naive I/O is easy to handle, one that does fancier I/O is harder.

4.1. The Standard Prestart Routines

If the user does not supply a prestart routine, IGOR supplies a default routine. It queries the user at the terminal to obtain the name of each file that should be opened, the mode for each file (read, write, etc.), and the position within the file at which the file pointer should start. It is thus possible to set the program's input/output system painstakingly to an arbitrary state, in particular to get back to the precise positions that the program was in at the time it stopped. It is also possible to move the pointers backward or forward to reprocess some data or to skip over some of it.

Another prestart routine is available to simplify the common case of programs in which all file output is sequential, and no file is rewritten. The names, modes, and file pointers of open files are dumped during each checkpoint; this information is used by the prestart routine to establish the environment. Gathering the information during checkpointing incurs additional costs, but the technique automates the restart process for many common scenarios.

4.2. User-Supplied Prestart Routines

Alternatively, the user may supply a specially crafted prestart routine to establish the I/O environment and perform any other desired actions prior to normal execution resumption. For example, if the user's program uses a screen manipulation package, the prestart routine might refresh the screen from the current memory image of the screen. A user-supplied routine may be worthwhile if a particular program is to be studied for a long time and will require frequent monitoring and restarts.

If the prestart routine does nothing, the environment at restart is the same as the environment for any normally executing program; i.e., the standard input, output, and error files are opened automatically, and refer to the terminal unless redirected to a file or a pipe.

4.3. More General Prestart Handling

In order to handle the general case, it would be necessary to keep a complete record of all input/output calls, those that control the unit and those that transmit data. The easiest way is to modify the library routines that interface to the system calls, but programs that circumvented the library would not be debuggable. A safe implementation would track all I/O requests in the kernel

5. REVERSIBLE EXECUTION

The functions described in the previous sections can support execution reversal. Two additional mechanisms, described below, have been implemented to enable us to reverse a program execution – a way of specifying the point to which execution will revert, and a method of reaching that precise point.

5.1. The Object Code Interpreter

In order to provide the necessary fine granularity in selecting a point in the execution, we implemented an object code interpreter for our system. Starting with the core image constructed from a checkpoint, the interpreter simulates the execution of the user's program forward from that point, stopping when the selection criterion is met (or when the program attempts to exit if the criterion is never satisfied).

5.2. Selection Criterion Specification

As described previously, the user can search the checkpoint data for the values of variables or arbitrary memory locations. Similarly the user might inspect the registers, the program counter, or the stack frames.

In the same way, we allow a user to specify the point to which execution is to be reversed by specifying a variable, a value, and a relational operator. For example, the user might wish to back up to the first point at which $x > 0$, or to the first point for which $n = 10$.

Ideally, the variable or variables in the search criterion should take on values that are monotone (either non-increasing or non-decreasing) during the course of the execution. This property allows IGOR to locate the last checkpoint before the desired point, from which the interpreter starts. (A variable whose value fluctuates up and down might satisfy a criterion at various times between checkpoints, while never satisfying it at any checkpoint.)

In the current implementation, selections must be based on a single comparison between a program variable (global, argument, or local) and a constant. A future version will permit the user to specify an arbitrary boolean combination of expressions, and to include references to the control flow in the program (with expressions such as *the program counter reached line 16 of function f*). A command that would find th first time interval after the thousandth line of a file had been processed might be

chfind 'lineno>1000'

6. PERFORMANCE CHARACTERISTICS

The typical edit-compile-test cycle might be modified for subsequent iterations by an edit-compile-reverse-restart cycle, with the use of these tools. We have measured the execution time and other performance criteria of the various tools used on several programs. The programs have not been carefully

tuned, and it is clear how to speed up some of the functions, but the performance is already quite acceptable for ordinary development use.

6.1. Performance of the Compiler

The compiler required almost no change to support the debugging tools. However, the compiler and linker must be run with options that retain extra symbol and relocation information and therefore run slower than in the standard mode. The execution time of the modified compiler was 17% greater than the standard compiler for a typical source file, and 37% greater for the compilation and linking of a 4700-line program. However, its performance was similar to that of the standard compiler when the -g debugging option is used (4% greater for compilation only; 11% greater for compilation and linking).

6.2. Characteristics of Checkpointing During Program Execution

We measured the performance of the checkpoint generation routines while executing three existing UNIX programs: *make*, the C compiler's main pass and *sort* with a 2853-line input file. These programs are, we hope, typical of those to be debugged, with reasonably simple but nontrivial input/output and a variety of memory access styles. The accuracy of this prediction can only be tested by real use. Table 1 shows the execution time overhead of the checkpointing (based on elapsed time).

Table 1: Execution Time Overhead

	Make		Ccom		Sort	
Checkpointing Interval (seconds)	Execution Time (seconds)	Overhead	Execution Time (seconds)	Overhead	Execution Time (seconds)	Overhead
none	4.7	–	12.1	–	22.2	–
1.0	8.0	70%	24.3	101%	31.7	43%
0.3	11.3	140%	34.3	183%	37.3	68%
0.1	22.7	380%	46.5	280%	47.5	114%

Table 2 indicates the average percentage of non-text memory pages that were written out during the execution. (Occasionally the total number of non-text pages accessed by the program appears to vary with different checkpointing intervals. At longer intervals, the checkpointing might not observe a peak in stack growth, or it might miss pages allocated between the last checkpoint and the program's termination. *Make* makes heavy use of allocated memory and linked lists, so a short execution has quite random memory access patterns.)

Table 2: Checkpointed Non-text Pages

Program	Checkpointing Interval (seconds)	Total Pages Accessed	Avg. Pages Written per Checkpt.	Percent Written
Make	1.0	46	34.0	74%
	0.3	47	27.9	59%
	0.1	47	23.9	51%
Ccom	1.0	89	41.2	46%
	0.3	89	36.5	41%
	0.1	89	30.7	34%
Sort	1.0	143	6.9	5%
	0.3	143	9.2	6%
	0.1	143	8.9	6%

6.3. Performance of the Incremental Loader

We tested the incremental loader by linking a replacement module into a 4700-line program consisting of nine modules. The incremental loader required 4.1 seconds of CPU time to link the replacement module and bind the variables to their original addresses. This represents an increase of 21% over the standard link edit run time of 3.4 seconds. (In addition, a total of 5.6 seconds were consumed by two supporting programs and a shell script that ties them together; most of this time would be eliminated by integrating the programs and eliminating the shell script.)

6.4. Execution Timing for Restart

In order to measure the overhead incurred in restarting a program execution, we ran *sort* on a file of 2853 records, which required 160 pages of memory. The execution time (in seconds) is indicated in Table 3.

Table 3: Restart Timing

	real time	user CPU	system CPU
Regular execution (with checkpointing on)	31.9	17.6	1.2
Restart	54.5	5.7	6.9
Portion of Restart in user program	10.1	3.9	0.3
Restart Overhead	44.4	1.8	6.6

In this example, the overhead of 1.8 seconds of user CPU time for restarting circumvents 13.7 seconds of user CPU time in the user's program (i.e., the amount of computation that preceded the restart point). The 6.6 seconds of system CPU time attributed to overhead would be decreased by integrating the several programs and the shell script involved in the process.

6.5. Execution Timing for the Interpreter

The time required by the interpreter to process an instruction mix of adds, comparisons, and branches is a factor of approximately 140 times slower than actual execution. Although not as fast as might be desired, it can be compensated for to some extent by decreasing the interval between checkpoints. In a later implementation, serious advantage might be taken of co-processors or other special hardware assistance to speed up the instruction stepping.

7. EXAMPLES

This section contains several examples that illustrate the use of IGOR. The commands typed by the user are shown in italic type, with the system's response in Roman type. Each example is based on a simple C program which converts the data in one or more input files to upper case.

7.1. Running a Program with Checkpointing

The program is compiled with the debugging version of the compiler. The user may optionally change the checkpointing interval (or turn off checkpointing); here a very short interval of 1 (hundredth-second) was used for illustrative purposes. The program is then executed in the usual way.

```
ncc  upcase.c -o upcase
chinter -o1 upcase                    # turn on checkpointing,
                                      #set timing interval to 1 * .01 sec
upcase input1 input2 >output          # checkpointing file names begin with R.
5056 characters were processed
```

7.2. Examining the History of a Variable

In the following example, the user inspects the value of the local static variable *chcount* in the function *main*. The output shows the checkpoint number, the execution time at the checkpoint (in real elapsed time, user CPU time, and system CPU time). Finally the value of the variable at the checkpoint is shown. In this example, most of the execution time was consumed in the buffer flushing portion of the standard I/O package, so most of the checkpoints occurred at times when the number of characters processed was a multiple of the buffer size.

```
chfind  main:chcount  upcase  # look for values of variable
checkpoint #  0 (r0.15 / u0.07 / s0.16)  value 1024
checkpoint #  1 (r0.23 / u0.09 / s0.20)  value 2048
checkpoint #  2 (r0.35 / u0.10 / s0.22)  value 3072
checkpoint #  3 (r0.37 / u0.11 / s0.23)  value 4096
checkpoint #  4 (r0.40 / u0.12 / s0.25)  value 4709
```

7.3. Backing up and Restarting

The program was then backed up to the point where the character count was 3500, and restarted from that point. The -q options provided for automatic handling of I/O reversal and restart, and directed file descriptor 1 (standard output) to the file named

```
reverse -q1output  "main:chcount >= 3500"  upcase
restart -q1output  -iR.core upcase
5056 characters were processed
```

8. FUTURE DIRECTIONS

We have shown here that general access to the execution history, including querying of that history, making changes, and restarting at an arbitrary point in an arbitrary state, can be provided in a general and efficient manner. The current system is just a prototype which needs further work.

8.1. Interfaces

To produce a fully useful debugger, the new capabilities should be integrated into a high-quality debugger that provides the usual facilities (menu access, symbolic listing of variables with certain properties, attractive displays, simple breakpoints). Examples of possible frameworks in the UNIX environment are the **pi** debugger [4] for Eighth Edition systems and **dbx** [5,6] for 4BSD systems.

8.2. Cleanups

The current system cannot search for system states based on general logical expressions or ones involving register variables. It ought to.

The general input/output case ought to be handled correctly, so that program state could be backed up with confidence even if files are modified randomly.

The tools are currently implemented as a collection of small standalone programs, which facilitated experimentation. Integration of the tools would eliminate some duplication and enhance their performance.

The interpreter could be sped up a great deal by using machine-dependent features. Some chips have facilities that permit single stepping of an execution, but usually only in kernel mode. If the interpreter could run in the kernel, it would run an order of magnitude faster, and thus time intervals of a few seconds would be tolerable.

8.3. Secondary Storage Reduction

The same checkpointing scheme would allow one to restart very long-running programs after a system failure. Less frequent checkpoints would be adequate (with an interval on the order of seconds or even minutes). It is easy to save a coarser time grid than was originally created by the checkpointer: it is only necessary to keep the most recent copy of each page changed during the coarser interval.

A special option to save only the most recent copy of each page would limit the amount of data accumulated yet automatically provide the advantages of a regular checkpoint facility.

Many programs change only a few cells on a page during a typical time interval. It would therefore often save a lot of disk space (at the cost of processor time) to store the differences in page values rather than the whole page. (As an aside, we found that around 1% of the pages that were written on did not actually change – they were the same at the end of the interval as at the beginning.)

8.4. Use of Multiple Processors

At present, the program to be debugged stops at each timer signal and copies its pages to disk. In a system with multiple processors and shared memory, a second processor could take up part of the load. The simplest implementation would copy the touched pages to shared memory, then move them (perhaps after compression) to disk. A more interesting implementation would make further use of the memory system: all data pages could be marked unwritable when the timer signals. A second process could then copy each page, then make it writable to the first process. If the first process tried writing on a page, it could make a copy of it then proceed. This technique is similar to the one used for garbage collection in [7].

8.5. Extensions to Asynchronous and Multi-Thread Programs

The current implementation assumes that all interesting events are directly caused by the running program. If external signals are important, it will be necessary to keep a record of them, with appropriate timing information, in order to permit true reversal of the execution sequence. It might be convenient to do a memory checkpoint at the times such external events occur, though it would still be necessary to trap periodically in the absence of signals.

A simple and valuable extension of this work would make checkpoints simultaneously on several cooperating processors. This could be accomplished by synchronizing the *ualarm* timer signals.

In a more general case, the ideas of the last two paragraphs would be combined to provide memory traces whenever an external or interprocessor event occurred or at synchronized time intervals.

REFERENCES

1) J. Conway, "The COPE User Interface", Proc of IEEE Compsac '82 (Computer Society 6th International Computer Software and Applications Conference, Chicago, 1982)

2) B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering* vol. SE-1, no. 2, pp. 220-232 (June 1975)

3) M. Pucci and J. Alberi, "The Architecture of the DUNE Multiple Processor System: An Experiment in Generalized Interprocessor Communication", Bellcore Technical Report

4) T. A. Cargill, "Pi – A Case Study in Object-Oriented Programming", Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, pp. 350-360 (September 1986)

5) M. Linton, "A Debugger for the Berkeley Pascal System", Master's project, UCB (June, 1981)

6) Sun Microsystems, Inc., "Debugging Tools for the Sun Workstation", Part No. 800-1325-03 (1986)

7) John R. Ellis, Kai Li, and Andrew W. Appel, "Real-time Concurrent Collection on Stock Multiprocessors", DEC Systems Research Center Technical Report #25 (February, 1988)

ACKNOWLEDGEMENTS

Marc Pucci implemented the *pagemod* and *ualarm* system calls in the operating system kernel, which enabled the efficient implementation of our checkpointing. Jim Alberi and Marc Pucci gave generous assistance in setting up and running our DUNE system.