An interactive software engineering tool for memory management and user program evaluation*

by WOLFGANG W. MILLBRANDT and JUAN RODRIGUEZ-ROSELL
Brown University
Providence, Rhode Island

INTRODUCTION

As the use of virtual memory becomes more and more accepted, the problem of effective storage management becomes more and more important. To date most efforts to optimize the use of memory have been directed at devising memory management strategies at the operating system level that minimize the number of page faults. For example, Comeau has shown that the loading sequence of subroutines can have a considerable effect on paging activity. Hence page-fault-optimizing loaders, linkage editors and compilers have been proposed. Although the concepts of “locality” and “working set” have been known for some time (c.f. Denning), little effort has been made to provide the programmer with suitable tools for making his programs “more local”. This seems to stem from the fact that, short of notions of “modular coding”, little is known about what sorts of programming habits actually result in local code. Consequently, most optimization techniques used to date have assumed that user programs were an unmodifiable input to the operating system.

Techniques for increasing locality of user programs and thereby reducing the paging overhead in a virtual memory environment have been investigated by Comeau, Hatfield and Ferrari. The methods presented involve the automatic restructuring of program modules into relocatable segments to increase the likelihood that page references that are close in time will also be close in space. In the Hatfield study the use of online displays to determine optimal segment sizes and view the effects of program reordering was found to be exceedingly helpful, but use of their system as an everyday programmer feedback and user program monitoring tool was never fully exploited.

The Brown University Display for Working Set References (affectionately known as BUDWSR) was primarily developed as a user-oriented tool to fill the need for user feedback systems by enabling the programmer to interactively monitor the memory referencing behavior of his modules. It was hoped that the programmer would be able to get a “feeling” of what it means to write localized code, and hence be able to modify his programming techniques in order to reduce, or at least have more control over, the memory resources required by his program. Although BUDWSR has not been in existence long enough for us to evaluate its effectiveness as a programmer training or feedback tool, we have already been able to establish some simple guidelines that allow the programmer to measure and hence increase the memory utilization of his programs. These guidelines could be used as a basis for establishing engineering standards for program evaluation. Furthermore, BUDWSR has proven itself to be an extremely powerful systems programmer tool that greatly facilitates the manual repackaging of modules in order to increase memory utilization.

SYSTEM ORGANIZATION

The system (c.f. Figure 1) essentially consists of a System/360 machine language interpreter and a satellite display processor. The user runs his program in much the same way as he would in a normal CP/CMS environment except that the interpreter counts the references made to memory pages (the page size is user defined) and periodically transfers its tables to the satellite computer. The interpreter runs in a 512K byte virtual machine and simulates a 256K CMS environment for the user’s program; the data gathered thus reflects only the activity of the user’s program and is not dependent on the virtual machine’s external environment. Since the System/360 instruction set is highly formatted, the basic data gathering facilities of the interpreter are fairly simple* and represent an acceptable cost.

* Most RR format instructions can be executed directly via the System/360’s execute instruction, while most RX, BS, and SI format instructions can simply be executed after computation of the base displacement address. Optionally, BUDWSR will not perform complete interrupt processing, e.g., interpretation stops when an interrupt is generated by the instruction stream, and resumes when control is returned from the interrupt handler. Since many of the CMS nucleus pages are in shared care, references to the nucleus code should not result in any significant paging overhead to the user.
The basic display (c.f. Figure 2a) produced by the satellite consists of page addresses plotted against time (measured in number of instructions). For each page address and time interval the system plots a small vertical spike whose height indicates how often that page was referenced during that time interval. At the start of the trace the user may define an arbitrary page size (from 8 bytes to 128K bytes) and time interval (from 1 to 65,535 instructions). Furthermore, he may ask the satellite to perform a non-destructive compression of the reference data, e.g., the user may ask the interpreter to gather statistics for 128 byte pages, and then view the results for 128, 256, . . . , or 128K byte pages. This allows him to view the programs behavior both on a global level (using for example a 4K byte page size) and to selectively “zoom-in” on any local peculiarities. To aid the programmer in identifying and restructuring inefficient modules, a cumulative reference count for each page, and module names and entry points (obtained from the CMS loader tables) are also displayed.

The satellite part of the system also uses the page reference data to produce a graph of the working set size versus time (c.f. Figure 2b). Thus the user not only has available which specific memory pages were referenced, but also a summary of the total memory used during a time interval. The working set size is computed in bytes for the user specified page size and also for the System/360-67 page size (4K). With a small page size (typically 128 bytes) the difference in the working set sizes indicates the amount of wasted space in the System/360-67 working set. Currently, we have measured only working set sizes and re-entry rates for an individual process. In reality, the amount of memory allocated to a process is a function of the paging algorithm and the load of the system. (Some data on system page fault and re-entry rates can be found in Rodriguez-Rosell and Dupuy.9) We have not concerned ourselves with simulations to determine page fault rates as a function of memory size or the operating system’s memory management policy. In general the amount of real memory available to the user is beyond his control; however, by using BUDWSR, the programmer can improve the locality of his programs while developing them, conceivably even changing to algorithms or data structures that might prove less noxious to the system. In return, he hopes that if his process uses memory efficiently the operating system will allow his process to execute rapidly.

The use of the satellite processor10 came about quite naturally in that we approached program monitoring as two
separate tasks, data gathering and analysis. We also wished to be able to monitor the output of the tracing program while it was running, rather than wait for off-line hard copy output of the execution record of the monitored program. Thus the satellite performs the multiple tasks of communication with the System/360 interpreter, updating the disk-resident master file containing the page references, and updating the display itself as new data are transferred by the interpreter or when the user "scrolls" through, or locally alters the page size for the memory display. The use of the satellite to process the trace data makes the system extremely flexible. Thus whenever we wish to determine whether or not a specific memory utilization parameter is meaningful, it is a simple matter to add code to the satellite program in order to calculate and display the parameter.

Another effect of using a satellite processor rather than a process running on the virtual machine itself is the fact that the interpretation and display data handling proceed in parallel in "real time." The interpreter runs unmolested by user interrupts for larger page sizes or different display lists. Thus it is possible to view a trace of a system, repackage it, and/or retrace it with different parameters all in the span of a few minutes. For monitoring an interactive system,** we can view a trace at the same time we are specifying commands to that system in order to quickly isolate and reduce the memory requirements for inefficient command modules. Since the trace data are immediately available after each time interval, it is also unnecessary to correlate the data from an entire run (such as would be given by an off-line plot) to the specific command or sequence of commands that generated the data. It is of course also possible to view previously taken trace data in a stand-alone mode of the satellite.

PARAMETERS FOR EVALUATION FOR PROGRAM MEMORY UTILIZATION

To date the system has demonstrated its usefulness through several practical applications. Many of the facts noticed about program behavior could of course have been predicted by use of common sense, and the reader may thus find some of the results rather unstartling. However, the fact that most of the observed traits are found in a large class of programs indicates that common sense rules (such as presented in the next section) are not used or even fully understood. It is hoped that by providing effective systems measurement tools and developing memory utilization standards, programmers will be made aware of their bad techniques (and also their good ones). Furthermore it has been found that a display is quite effective in pointing out some of the parameters that may be used to define standards

* In general, the information to be displayed is larger than the available display area (e.g., the CRT display can effectively represent 50 4K-byte pages, but not 800 256-byte pages).
** For example, some of this paper was edited under a text editor being monitored (see below).
trace of the FORTRAN F Compiler in Figures 2a and b). Consistently unstable working sets are indicative of poor process behavior.

The second component of the working set entry rate, we will attribute to page replacement rate. It is a measure of the change in the membership of the working set as a result of single pages entering or leaving the working set. Thus programs with a nearly constant working set size may still have a non-zero re-entry rate due to page replacement within the working set. This component of the re-entry rate was found to be meaningful only for programs with large (>40K) working sets. Program references to infrequently used data areas or error routines seem to result in a small set of “fringe” pages that wander in and out of an otherwise stable working set. Little if any page replacement was observed for programs with small working sets.

It should be noted that individual parameters are quite sensitive to the time interval used in collecting the data. Thus it is quite possible to use a small time interval to artificially reduce the size of the working set. This is however offset by the fact that if a program is using memory inefficiently, reducing the time interval will correspondingly increase the re-entry rate of new pages into the working set. Conversely, although a long time interval will increase the size of the working set, it will also decrease the re-entry rate. The time dependencies of the parameters can be used quite advantageously in the process of packaging system modules. Thus we can first choose a long time interval and package to reduce the total size of the working set as much as possible. We can then use a small time interval to repack the heavily used routines found by the first packaging into a stable working set configuration. After the initial packaging iterations it is still possible to fine tune a user system by appropriate time interval selection. Thus if we are optimizing an interactive text editor we might select a time interval that spans a single editing function; if we are optimizing a CPU bound applications program we might choose a time interval that corresponds to the operating system’s time slice, etc.

SOME CASE STUDIES

FRESS—A file retrieval and editing system

FRESS is a sophisticated interactive text handling system. It has run in a 120K MVT partition (using dynamic loading), but in the CP-67 environment it was found more convenient to relinquish the memory management to the virtual paging system. Even though FRESS is a highly modular system, and entirely code in assembly language, it was designed and implemented without paging load criteria in mind. Thus it was not too surprising that the FRESS users were typically the first to be impacted at times when Brown’s 67 neared saturation.

The following observations of the unpackaged (e.g., as loaded by the CMS loader) version generally accounted for this behavior:

- The working set size was rather large, roughly 128K (with a 4K page size and a time interval of 5,000 instructions).
- The working set was extremely unstable, i.e., the working set size would vary from 60 to 128K between any two user activated functions.
- And finally, the amount of the CP-67 working set actually used during any one time interval was typically around 30 percent.

Fortunately it was possible to remedy most of the above characteristics by simply reloading the system. Thus after noting which modules were frequently in use, and which were not, and hand specifying the module loading order, the following results were obtained:

- The working set size dropped to 80K (a reduction of 30 percent)
- The working set size became more stable, generally changing by less than 16K for the common editing functions.
- The page utilization figure increased to 50 percent.

Even with the increased memory utilization achieved by a repackaging of the FRESS system, further reductions of another 20K in the size of the working set still seem possible by breaking up some of the larger modules and alignment of some data areas on 4K page boundaries (some guidelines to enhance the “packageability” of programs are presented in the next section).

The CSS editor

The CSS editor is a line oriented text editor that is provided as a user service on the NCSS time sharing service. Since NCSS’s Duplex 67 must frequently support many users the following statistics should not be too surprising:

- The working set size was a small 12K±4K (with a 4K page size).
- The working set was very stable due to the fact that most of the commonly used functions had been loaded onto the first CMS user memory page.
- The memory utilization fluctuated from 30 to 50 percent depending on the user function. The lower memory utilization figures were usually due to editing functions that generated data references across the 4K page boundaries. Since many fixed length line data files contain about 40 percent blanks, one wonders how much the utilization factor would increase if some data compression were built into the editor.

* In particular some modules, such as the command language interpreter, are executed for every command; some, such as the various editing functions are seldom invoked at the same time, but do share some common subroutines, etc.
SCRIPT AND NSCRIPT

SCRIPT and NSCRIPT are text formatting programs that can be used in conjunction with the CMS or CSS text editors. NSCRIPT is an MIT version of SCRIPT that supports some more advanced features such as footnote placement and user macros. For purposes of comparison we monitored both programs while they processed the same input file (obviously none of the extra features provided by NSCRIPT were included in the file). We let the user draw his own conclusions from a comparison of the following data:

- The SCRIPT working set size was 8K.
- The working set was stable.
- Memory utilization ranged from 50 to 70 percent.

- The NSCRIPT working set size was 24K.
- The working set was stable.
- Memory utilization ranged from 30 to 50 percent.

FORTRAN F Compiler

The Figures 2a and b are representative of any particular Fortran compilation, e.g., changing the source program to be compiled may change the duration of each phase but will not significantly change the characteristics of the two graphs (here we are assuming that no exceptional conditions such as error messages occur). The page size for Figure 2a is 2K bytes (for a more detailed description of the figures see the first part of this paper). The plots in Figure 2b are working set size with a 4K page size (top line), working set size with a 128 byte page size (center line), and entry rate for 4K pages (bottom line). The time interval for both figures is 10,000 instructions.

Some CPU simulation programs

One of the earlier demonstrations of our system was for some representatives of a systems measurement group at the Mitre Corporation who brought two of their programs for observation on our system. The first was a Fortran program which had been analysed as to the number and kinds of source statements it contained. The second program (also written in Fortran) accepted the analysis data as input and simulated the "resource usage" behavior of the first program. We were to observe the behavior of both programs to see how good a job the simulator program was doing. Although the CPU load of the two programs could have been considered to be equivalent, we observed with BUDWSR that their memory utilization patterns were in no way related to each other. This might serve as a reminder that program behavior is still poorly understood, and that a good deal of empirical data gathering might still be in order.

SOME PRACTICAL GUIDELINES TO INCREASE MEMORY UTILIZATION

Many of the suggestions in this section can also be found in Morrison. We have presented here those situations that have been observed to have the greatest potential for reducing the memory requirement of a program. Most of these observations are applicable only to the instruction stream of the monitored program, and not to the data references. We are currently experimenting with monitoring only data referencing (read and/or write) patterns in order to be able to understand some of the finer details of memory management. It should also be noted that a good systems programming tool, such as BUDWSR, is an invaluable addition to the commonly used guidelines.

- System modules should be as small as possible since this greatly facilitates global repackaging.
- A module should execute as much of its code as possible when it is called. Hence special cases and error conditions should be diagnosed inline, but handled by calls to separate modules.
- Large initialization sections should be handled as calls to separately created modules if necessary. This has the effect of compacting the more frequently used code and also allows the programmer to group together the various initialization sequences. The most common case of inefficient memory usage found by our observations was the first 128 to 512 bytes of initialization code in a module, that were executed only at entry to the system.
- Quite frequently the packaging of subroutine libraries used by higher level languages was not done with virtual memory in mind. Thus the system programmer should be especially cautious of some unnecessary overhead brought about by inappropriate loading of run time routines. For example the following PL/I statements generate a 16K to 24K working set in our CMS environment (note that this figure does not include the pages used by the CMS I/O handler):

```
TEST: PROC OPTIONS(MAIN);
DCL I BIN FIXED;
DO I=1 TO 10;
   PUT LIST (I);
END;
END TEST;
```

- Careful consideration should be given to user management of his data structures, i.e., it is often preferable to allocate a few large chunks of memory, rather than many small ones, so that data references may be organized in an efficient manner. The same holds true for data organization using the Fortran COMMON statement.
- Heavily used routines and frequently referenced control blocks or I/O buffers should be allocated in an order...
such that they do not cross page boundaries unnecessarily.

- Software implemented stacks for a subroutine’s save area and local variables may be of considerable use in efficiently organizing data references.
- Error message text and error message handlers should be grouped separately from normal flow of control. Frequently used “prompt” messages should of course be kept separate from the error message modules.
- The use of literal pools at the end of large programs (or more typically at the bottom of the first 4K of a program) should be avoided. Better yet, literal data should be treated as part of the instruction stream and placed as soon as possible after its use.

CONCLUSIONS AND FUTURE WORK

It is clear from our experience that the use of on-line displays and satellite processors as measurement tools is far more flexible than use of batch-oriented measurement systems. Satellites will be increasingly used to monitor and assist both the operating system and user software. From an operating system point of view, it is expected that monitoring processors will be attached to large mainframe CPUs as an integral part of the operating system (as is done with the CDC STAR-100). Thus we hope that as the larger systems become more complex, data gathering facilities (such as page reference counts and general paging activity) will be implemented in the microcode of the mainframe CPU and be directly available to the satellite for processing, thus bypassing the need for system measurement via interpreters, simulators and the like. Just as today, the mainframe relinquishes I/O operations to the channel, future mainframes may pass on user behavior data (by means of control store) to a satellite that will compute new working set size parameters, while the mainframe processes another user. Furthermore, this arrangement makes the behavior of the mainframe system insensitive to the data analysis complexity, as well as allowing modification of the data analysis by a reprogramming of the satellite.

From a user software point of view, it is expected that some standards (for CPU and memory utilization) will be established for programs in production use. Thus the fact that by a simple repackaging it is frequently possible to increase the memory utilization by 20 to 30 percent may indicate that quantitative rules such as “use 50 percent or more of the memory allocated,” could be established. These rules, however, will be of little use if we do not provide adequate measurement tools, such as BUDWSR, that allow the user to monitor his programs.

In view of the recent technological developments, it is very possible that future computer systems will have very large random access memories, possibly of the order of 50 megabytes. A new dimension is then added to the evaluation problem, for then the determining factor will be data locality, rather than program execution locality. Programs will fit entirely in memory, but the larger data base oriented systems of the future will certainly need techniques to increase the data locality at all levels in the data base hierarchy. Engineering tools are necessary to evaluate and control solutions to user requirements.

REFERENCES