November 30, 2000

SRC Research Report **167**

A Practical Approach for Recovery of Evicted Variables

Caroline Tice
and
Susan L. Graham

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

A Practical Approach for Recovery of Evicted Variables

Caroline Tice and Susan L. Graham

November 30, 2000

Professor Susan L. Graham works in the Electrical Engineering and Computer Science Department at the University of California, Berkeley. She can be reached by email at graham@cs.berkeley.edu.

Abstract

The *residency problem* arises in the context of debugging optimized code: a user may ask to see the value of a variable that is currently in scope, but whose value has been overwritten (evicted) and is therefore not available. This creates a difficult situation for the debugger. On the one hand, allowing users to examine the state of programs as they execute is the raison d'être for having debuggers. On the other hand, the debugger cannot return a value if the value is not in the computer. Because the residency problem is very common in optimized code, it is critical that any debugger for optimized code have some mechanism for dealing with it appropriately.

There are two parts to this problem: detecting that the variable has been evicted and is no longer resident; and responding appropriately to the user's request for its value. Although residency determination has been studied before, recovery of evicted values apparently has not. In this paper we present Limited Eviction Recovery, a simple, practical solution for the residency problem. In this solution the compiler informs the debugger of those locations in the binary where source variables are evicted, and the debugger makes copies of the values of the variables before they are overwritten and lost. In order to avoid incurring too much cost, the debugger only preserves the values of the variables most likely to be of interest to the user. In addition to explaining our algorithm, this paper describes our implementation of Limited Eviction Recovery, presents measurements taken of the resulting system, and discusses some implications thereof.

Contents

1 Introduction

One of the most difficult tasks that a debugger for optimized code must perform is to find and return the value of a source variable in response to a user's query. There are three main problems that contribute to this difficulty: the currency problem, the data location problem, and the residency problem. Collectively these are known as the *data value problems* [5]. The *currency problem* is detecting and recovering from situations where the actual value of the variable is different from the value the user would expect based on examining the source program. Such situations can arise when optimizations change the order in which variable assignments occur. The user, on discovering the actual value differs from the expected, may be misled into thinking there is a bug when there is not one. The *data location problem* is determining the correct location (register or memory address) in which to look for the current value of a variable. This can be difficult because optimizations frequently copy and move variables and often eliminate the store of the variables' values to memory. The *residency problem* is recognizing and coping with the case where the requested variable's value is not currently stored *anywhere* in the computer and is therefore not available. In this paper we focus on the residency problem. Approaches for handling the currency problem and the data location problem are given elsewhere. [2, 3, 4, 5, 6, 7, 9, 12, 13, 14, 15]

The residency problem can arise because a variable's lifetime is not necessarily the same as its scope. At any point in the execution of a program, a variable which may be within the current scope can also be "dead", i.e. its value is no longer needed for the computation. In such cases, the optimizer allows the variable's storage location to be re-used for some other value without first saving the value of the dead variable anywhere. From the compiler's point of view this is not a problem, as the dead value is no longer needed for the computation. From the debugger's point of view this is unfortunate, because the user may wish to examine that value anyway. Whenever a variable's value is overwritten without first being saved elsewhere, the compiler is said to have *evicted* the variable, and the variable is said to have become *non-resident*.

If variable evictions were rare, it might suffice to detect those few places where they happened, and to inform the user in such cases that the variable's value is unavailable. However, variable evictions are widespread in optimized programs. In a study performed by Adl-Tabatabai and Gross [1], they found that 30–60% of all possible breakpoint locations had variables within scope that were evicted. Because variable evictions are so common, it is imperative for debuggers of optimized code to properly address them.

In this paper we present Limited Eviction Recovery, a practical approach for recovering the values of evicted variables in most situations. The rest of this paper is organized as follows: Section 2 presents and discusses general techniques for handling evicted variables in a debugger. We present our algorithms and solution in detail in Section 3. In Section 4 we describe an implementation of our solution. Section 5 presents and analyzes measurements of our implementation, and our conclusions are presented in Section 6. We know of no prior work on eviction recovery, although Adl-Tabatabai and Gross [1] have presented a technique for detecting evicted variables.

2 Potential Solutions for the Residency Problem

There are three basic approaches for dealing with evicted variables: detection, reconstruction, and value capture.

Detection is the simplest method. It involves merely keeping track of which variables are evicted at each location in the target program. In response to a request to see a variable's value, if the variable is evicted, the debugger can inform the user that the variable's value is not currently available. As previously mentioned, this is the approach taken by Adl-Tabatabai and Gross [1]. While relatively easy to implement, this approach is not likely to be satisfactory. Given the frequency with which eviction occurs, it is probable that a large number of user requests would receive such a response, which in turn is likely to frustrate the user.

Reconstruction attempts to recompute the last value of an evicted variable from other values, in the same way that that value was first computed. There are several problems with this approach. To begin with, keeping track of all the components required to reconstruct each variable in the program requires recording a huge amount of additional data. Second, even if that data were collected and passed to the debugger, it is possible that one or more of the contributing variables will have been updated since the evicted variable received its value. In fact it is likely that some of the operands themselves have been evicted, and would need to be reconstructed. The sheer volume of information required by this approach makes it impractical.

In value capture, which is the approach advocated in this paper, the debugger notices when a variable is about to be evicted and captures the value of the variable before the eviction takes place. The debugger stores these saved values in a special list, which it uses to lookup the values of evicted variables in response to user requests. We discuss this approach more fully in the next section.

3 Limited Eviction Recovery

The obvious way for a debugger to capture values just before eviction is to use *hidden breakpoints* [15]. Suppose that a variable is about to be evicted. In order to copy the variable's value before it is overwritten, execution of the target program must be suspended prior to the eviction and control given to the debugger. Therefore the debugger must set breakpoints on all instructions that evict variables. Furthermore these are not user-specified breakpoints; since the debugger does not pass control to the user at these positions, they are *hidden breakpoints*.

While these breakpoints are hidden from the user, they are not free. Every time a hidden breakpoint is reached, the kernel must switch contexts and give control to the debugger, which will collect the appropriate value and return control to the kernel, which then resumes executing the target program. Given the high frequency of variable eviction, using hidden breakpoints (each of which requires two context switches) to save the value of every variable that is about to become evicted will probably have a highly visible, negative impact on the running time of the program. Thus there is a conflict between needing the ability to present evicted variables' values to the user and keeping the impact on the execution speed of the target program to a minimum.

To address this cost issue, we introduce Limited Eviction Recovery. In Limited Eviction Recovery, the debugger uses hidden breakpoints to copy the values of evicted variables, but only in those subroutines where the user has already set a control breakpoint, and only while the control breakpoints are active. Thus, while the running time for those subroutines will be affected by the eviction recovery scheme, the rest of the program will not. Since, by choosing to set a breakpoint in the subroutine, the user has already indicated a willingness to slow down (stop) execution time within the subroutine, this tradeoff seems fair. In fact our measurements indicate that, in general, the time spent servicing hidden breakpoints and recovering evicted variable values is completely overshadowed by the time spent servicing the user's control breakpoints.

The benefit of this approach is that, whenever execution of the target program stops at a user-specified control breakpoint, the values of all variables local to that subroutine will be available for the user to examine. The values of global variables and of any arguments to any functions on the call stack will also be available for examination, since they are normally assumed by the compiler to be live, and live variables cannot be evicted. Thus the values of the variables the user is most likely to be interested in are available for examination, while at the same time, by limiting eviction recovery to subroutines where the user has set active control breakpoints, the impact of eviction recovery on program run time is minimized.

Obviously this approach also has some limitations. Target program execution can be suspended for reasons other than a user-specified control breakpoint (e.g., a data breakpoint, a user interrupt, or a fault). In such cases, eviction recovery may not have been performed, so variables within the current subroutine may or may not be available. Another problem is that, when moving up or down the call stack, variables that are *not* arguments to the function calls may or may not be available. Similar problems also exist for debugging core dumps (post-mortem debugging). In those situations where a user requests the value of an evicted variable for which the debugger does not have a saved value, we fall back on detection, informing the user that the requested value is unavailable.

4 Implementation Details

In order to test our ideas, we implemented Limited Eviction Recovery in Optdbx. Optdbx is part of a suite of tools created to allow accurate, truthful debugging of optimized code. These tools consist of a slightly modified version of the MIPS-Pro 7.2 C compiler, a commercial compiler developed by Silicon Graphics, Inc. (SGI) that optimizes aggressively; Optview, a tool which takes a C source program and generates an optimized version of the source, based on the optimizations the MIPS-Pro 7.2 C compiler performed when it compiled the program; and Optdbx, a modified version of the SGI dbx debugger. More complete descriptions of these tools can be found elsewhere [10, 11]. Here we will focus on those portions relevant to eviction recovery.

We modified the compiler to perform dataflow analysis on the final version of the optimized binary in order to discover the correct locations in which to look for the values of all source variables. This was necessary for solving the data location problem [9]. In the process of doing this dataflow analysis, the compiler also discovers all the places in the target program where variable eviction occurs. The compiler records information about each eviction location in a special portion of the symbol table, which Optdbx later uses. We use the DWARF 2.0 symbol table format [8], creating a new, company-specific section in the table, as allowed by the DWARF specification, to contain the variable eviction information.

There are three parts to performing Limited Eviction Recovery in Optdbx: setting up or discontinuing eviction recovery within a subroutine; servicing the hidden breakpoints; and servicing user requests for evicted variables. We will discuss each of these in turn.

In order to set up or discontinue eviction recovery, Optdbx keeps a counter for every subroutine in the executing program. This counter tracks the number of control breakpoints the user currently has set in the subroutine. Every time the user sets a breakpoint in a subroutine or chooses to step into the subroutine, Optdbx checks the subroutine's breakpoint counter, and increments it. If the requested breakpoint is the first breakpoint the user has set in the subroutine, Optdbx sets up eviction recovery within that subroutine (as described below). Similarly, every time the user removes a control breakpoint or chooses to step out of a subroutine, Optdbx decrements the subroutine's breakpoint counter. If the breakpoint counter goes to zero, Optdbx discontinues eviction recovery within that subroutine. To set up eviction recovery within a subroutine, Optdbx checks the symbol table to find all the addresses within the subroutine that evict variables. It then sets a hidden breakpoint at each of those addresses. To discontinue eviction recovery within a subroutine, Optdbx removes these hidden breakpoints.

During execution of the target program, if the kernel encounters one of the hidden breakpoints it suspends execution of the program and passes control to Optdbx. Optdbx checks the symbol table to see which variable is about to be evicted and then copies the variable's value from its current location to a special list Optdbx maintains.[1] Once the value has been captured in this way, control is returned to the kernel, which resumes executing the target program.

Any time the user requests a variable's value, Optdbx consults the symbol table, using the current program-counter value, to see where the variable is presently stored. If the symbol table indicates that the variable is currently evicted, Optdbx looks up the variable's name in its special list and checks to see if there is a corresponding value. If the value is in the list, Optdbx returns the value. Otherwise Optdbx reports to the user that the variable's value is currently unavailable. As previously mentioned, this last case can occur if execution has been suspended for some reason other than a control breakpoint, or if the user has changed frames in the execution stack, and the program is in a subroutine for which eviction recovery was not enabled.

The MIPS-Pro 7.2 C compiler consists of roughly 500,000 lines of C and C++ code. Modifying the compiler to perform the dataflow analysis in order to detect all the locations at which source variables are evicted took roughly two months, and required writing or modifying roughly 1,500 lines of code. Adding a new section to the symbol table to record the eviction information involved writing about 500 lines of code and took 2-3 weeks.

The SGI dbx debugger, on which Optdbx is based, consists of roughly 150-200,000 lines of C++ code. Modifying the debugger to count the control breakpoints in each subroutine, to set and remove the hidden breakpoints when appropriate, to determine which variable is about to be evicted when a hidden break-

[1] A separate list is actually maintained for each stack frame for each subroutine, in order to handle recursion correctly. Optdbx stores the list with the other information it maintains about each stack frame.

	Time w/o Recovery	Time w/ Recovery	Slowdown secs. (%)	# Static Traps	# Dynamic Traps	Avg. Time/ Recovery
go	666	706 (4 funcs)	40 (6.0%)	37	1,614	0.025
compress	784	788 (1 funcs)	4 (0.5%)	19	552	0.007
li	566	799 (1 funcs)	233 (41.0%)	12	21,600	0.011
ijpeg	270	377 (6 funcs)	107 (39.6%)	64	8,136	0.013
perl	309	323 (7 funcs)	14 (4.5%)	74	590	0.024
philspel	1	122 (3 funcs)	121 (12,100.0%)	16	9,940	0.012
eval-emon	1	97 (4 funcs)	96 (9,600.0%)	570	612	0.168
measure1	9	1,985 (1 funcs)	1,976 (21,955.6%)	149	78,204	0.025
Average	–	–	–	–	–	0.036

Table 1: Eviction recovery timing measurements (in seconds)

point is reached, to find and copy the variable's value, and to return the appropriate value when the user requests an evicted variable required writing or modifying approximately 1,000 lines of code and took about two months. Thus the total effort involved in implementing Limited Eviction Recovery required writing or modifying about 3,000 lines of code and took 4.5 months. The amount of work would have been less if the implementer had already been familiar with the compiler and debugger code. These numbers show that Limited Eviction Recovery can be implemented within existing compilers and debuggers with only a moderate amount of effort.

5 Measurements and Analysis

The reason we have given for choosing our approach, rather than performing eviction recovery throughout the entire program, was the claim that attempting the latter would have too much of a negative impact on the execution time of the program, while the solution we chose would not.

To verify this claim we took several different timing measurements within the debugger. These measurements were taken by running benchmark programs from within the debugger, using various levels of eviction recovery. We modified the debugger to perform eviction recovery in selected functions, without setting any control breakpoints, since such breakpoints would interfere with our timing measurements. The time it took each program to execute was measured. As the most important question to answer is not how long it takes the CPU to execute the code, but how long the user has to wait for the program to complete, the time measured was the total elapsed time, from the user's perspective. This time was obtained by modifying the debugger to print the current system time both when it initially starts executing the program to be monitored, and when the execution of the program terminates. We then took the difference between these two times.

The programs we used as benchmarks include five of the C programs from the SPEC95 benchmark suite (`go, compress, li, ijpeg and perl`). Since we were concerned about the possibility that the SPEC benchmark programs might have characteristics that are not representative of other C programs, we also measured some C programs written by graduate students at the University of California, Berkeley. These programs are `philspel`, a small program that takes a dictionary and a document and performs spell-checking on the document; `eval-emon`, a 5,000 line program that takes as input files containing data from hardware counters, and performs a series of calculations on this data, determining such things as cache miss rates; and, `measure1`, a group of simulation scripts for testing a network resource clustering algorithm. The results of our eviction recovery measurements on these eight benchmarks are shown in Table 1. All the times shown are in seconds.

The first column of numbers shows the time it took to execute the program without any eviction recovery. The second column shows the execution time with eviction recovery being performed in one or more functions. The number of functions with eviction recovery is shown in parentheses. The third column shows the difference between the first two columns, indicating the time spent on eviction recovery. The actual slowdown in seconds in shown first, followed by the percentage slowdown in parentheses. The next two columns show the number of hidden breakpoints set (static) and the number of eviction recoveries performed during execution (dynamic). The final column shows the average time spent on each eviction recovery. This varies anywhere from 7 milliseconds to 168 milliseconds, with the overall average being 36 milliseconds.

The reader should note that these are worst-case measurements. For simplicity, we implemented the hidden breakpoints using context switches, which are expensive. It would probably be possible to speed up eviction recovery significantly by using code patching instead. Also, since we measured eviction recovery in selected subroutines without setting and removing control breakpoints, we perform eviction recovery in each selected subroutine *every* time it is called, rather than on user request. This is not the way in which we expect Limited Eviction Recovery to be used.

Not surprisingly, the largest percentage slowdown in our measurements occurred in the three programs with very short "normal" execution times (two of the programs took one second or less to execute without eviction recovery). In seven out of the eight benchmarks, the additional time spent on eviction recovery ranged from a few seconds to less than four minutes. At first glance this slowdown may seem a heavy penalty to pay for eviction recovery. Indeed it confirms our original hypothesis that performing full eviction recovery everywhere would cost too much in terms of execution time. In fact it would be easy to erroneously conclude from these numbers that even our partial eviction recovery is too expensive and impractical. But such a conclusion overlooks a critical aspect of our limited eviction recovery scheme: eviction recovery is only performed *while the user has breakpoints set in the subroutine*.

The following example should help illustrate the importance of this point. While looking for appropriate subroutines in which to set up eviction recovery for our measurements, we came across one case in the SPEC benchmarks where a single function was called over 100 million times during execution. However for each individual call to the subroutine only 30 eviction recoveries were performed. At an average of 0.036 seconds per eviction recovery, performing eviction recovery in this subroutine would add an additional 1.08 seconds to the execution time of the program, each time the subroutine is called with eviction recovery activated. Recall that in our scheme eviction recovery should only be activated in this subroutine while the user has an active control breakpoint set there. Since obtaining the user's request at the breakpoint and servicing it will take far longer than 1.08 seconds, it is fairly safe to state that, for any given invocation of the subroutine, the time spent performing eviction recovery will be completely overshadowed by the time spent servicing the breakpoint, and will probably pass unnoticed. It is also likely that the user will only leave the breakpoint active in the subroutine for a tiny fraction of the 100 million times the function is called. For the rest of the subroutine invocations, when the breakpoint is not active, eviction recovery will not be performed and the subroutine will execute at its normal speed. However if we performed eviction recovery in the subroutine every time it is called, which is the methodology we used for collecting our measurements in Table 1, it would take an additional 3.4 *years* to execute. Admittedly this particular example is a little extreme, but it does show the importance of limiting eviction recovery to only those subroutine invocations for which an active control breakpoint is set. Although full eviction recovery is not practical for this subroutine, Limited Eviction Recovery is.

6 Conclusions

Optimizing compilers often produce code that evicts variables while they are still in scope. This poses a major problem for debuggers when users ask to see the values of such variables. A debugger can prepare to answer such user queries by saving the variable values at the time of eviction. However doing this requires the use of hidden breakpoints, and using hidden breakpoints for *all* evictions can be slow. We have therefore proposed Limited Eviction Recovery, a scheme which reduces the performance impact by selecting only the most critical evicted values for saving: the values evicted during the execution of a subroutine in which the user has set control breakpoints. We have implemented this scheme and we have shown that, although performing full eviction recovery might be too costly, in terms of the effects on program execution time, performing limited eviction recovery is practical and greatly improves the information available to users about the values of variables.

References

[1] A. Adl-Tabatabai and T. Gross, "Evicted Variables and the Interaction of Global Register Allocation and Symbolic Debugging", *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993, pp. 371-383.

[2] A. Adl-Tabatabai, "Source-Level Debugging of Globally Optimized Code", Ph.D. Dissertation, Carnegie-Mellon University, CMU-CS-96-133, May 1996.

[3] M. Copperman and C. McDowell, "Technical Correspondence: A Further Note on Hennessy's "Symbolic Debugging of Optimized Code" ", *Transactions on Programming Languages and Systems* 15(2), April 1993, pp. 357-365.

[4] M. Copperman, "Debugging Optimized Code without Being Misled", Ph.D. Dissertation, University of California, Santa Cruz, May 1994.

[5] D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", In *Proceedings of the 1988 PLDI Conference*, 1988.

[6] J. Hennessy, "Symbolic Debugging of Optimized Code", *Transactions on Programming Languages and Systems* 4(3), July 1982, pp. 323-344.

[7] W. Shu, "Adapting a debugger for optimised programs", *ACM SIGPLAN Notices* 28(4), April 1993, pp. 39-44.

[8] J. Silverstein, ed., "DWARF Debugging Information Format", Proposed Standard, UNIX International Programming Languages Special Interest Group, July 1993.

[9] C. Tice and S. Graham, “A Practical, Robust Method for Generating Variable Range Tables”, Research Report 165, Compaq Systems Research Center, Palo Alto, CA, September 2000.

[10] C. Tice, “Non-Transparent Debugging of Optimized Code”, Ph.D. Dissertation, Tech. Report UCB//CSD-99-1077, University of California, Berkeley, Oct. 1999.

[11] C. Tice and S. Graham, “OPTVIEW: A New Approach to Examining Optimized Code”, *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, ACM SIGPLAN Notices 33(7), July 1998, pp. 19-26.

[12] D. Wall, A. Srivastava, and F. Templin, “Technical Correspondence: A Note on Hennessy's “Symbolic Debugging of Optimized Code” ”, *Transactions on Programming Languages and Systems* 7(1), January 1985, pp. 176-181.

[13] R. Wismüller, “Debugging of Globally Optimized Programs Using Data Flow Analysis”, *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 29(6), June 1994, pp. 278-289.

[14] L. Wu, R. Mirani, H. Patil, B. Olsen and W. Hwu, “A New Framework for Debugging Globally Optimized Code”, in *Proceedings of the ACM SIPLAN '99 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 34(5), May 1999, pp. 181-191.

[15] P. Zellweger, “High Level Debugging of Optimized Code”, Ph.D. Dissertation, University of California, Berkeley, Xerox PARC TR CSL-84-5, May 1984.