Ideas for Efficient Hardware-Assisted Data Breakpoints

Jonathan E. Cook   Mayur Palankar
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

Data breakpoints, sometimes called watchpoints, have long been desirable for debugging and other dynamic analyses, but are often prohibitively slow to use. Current processors have a small number of breakpoint registers that can be used to trap data read and write operations at CPU speeds—for example, the Intel 386+ CPUs have four breakpoint registers that can watch one word of memory each. Current use of these registers is naive and limited, and so we propose and describe some investigation into furthering their use.

1. Introduction

While debuggers have long supported efficient code breakpoints, data breakpoints, sometimes called watchpoints, have lagged behind in the efficiency. This is because debuggers have typically resorted to single-stepping through the program and checking to see if the current instruction is going to touch the watched data location. Code breakpoints are easy because there is only one place in the code to worry about, and a trap can easily be set at that point. A data location can be used or assigned in many code locations, and in a program with pointers it is possibly undecidable as to which code locations will affect a specific data location. One reference cites a slowdown of 85,000 times for a program running under a debugger with a data watchpoint set [3].

Current processors have attempted to alleviate this situation somewhat by including in their design a small number of breakpoint registers that can be used to trap data read and write operations at CPU speeds—for example, the Intel 386+ CPUs have four breakpoint registers that can watch one word of memory each. Other CPUs have just one data breakpoint register.

These registers are a step forward but are obviously a severely limited resource. For example, the *gdb* debugger will use the breakpoint registers for simple variable access breakpoints, but will resort to software trapping if more data is being watched than there are registers, or if expressions are used. For example, the program

```
int main()
{
   int x,y,z;
   y = 5; z = 3;
   for (x=0; x<10000000; x++)
   {
      if (x == 678456)
         y = 2;
      z = x - 5;
   }
   z = (x-z) + y;
   return z;
}
```

when run directly gives an execution time of 0.04 seconds. Running it under *gdb* with no watchpoints gives an instantaneous prompt return (meaning essentially no slowdown), and running it with a “watch y” also gives an instantaneous program interruption at the “y=2” line. Unfortunately, running the program while watching for the expression “y==5” to change takes 6 minutes, 15 seconds in the debugger process and 41 seconds in the program process. This gives a total slowdown of about 10,400.

Thus, while the data breakpoint registers are being used currently, their use is basic and naive. We propose several interesting research questions surrounding the use of these registers. Can these limited breakpoint registers be used efficiently to watch a large number of data locations? If so, what types of extra support are needed to be able to schedule the registers? Can static analysis of the program help determine the scheduling of which locations needs to be watched at each point in the program runtime? If 100% coverage is too expensive to obtain, can statistical methods be used to achieve high but not perfect coverage?

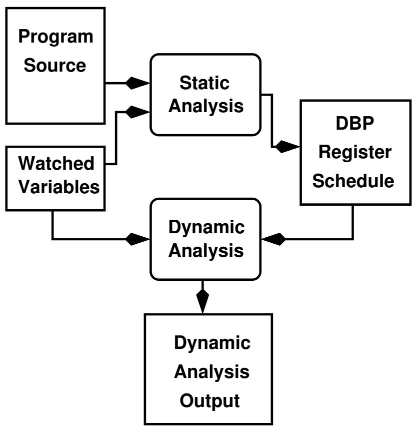

Figure 1. General framework for breakpoint register usage.

In this paper we explore some not-yet-tested thoughts on how these data breakpoint (DBP) registers might be used for efficient dynamic analyses that need to watch variables. Figure 1 shows our general framework for efficiently using the DBP registers in dynamic analyses. We propose that a static analysis phase is needed to build an efficient (or even feasible) schedule when trying to use limited DBP registers to watch a large number of variables.

2. Assumed instrumentation capabilities

Our goal is to improve the efficiency of dynamic analyses that need to watch variables. Thus, while in the extreme we might need full instruction-level instrumentation capabilities, our goal is to use as little instrumentation as possible. The two questions regarding watching more variables than are DBP registers are 1) Does a schedule for the DBP registers exist that covers all accesses to the variables, and 2) What triggers are needed in the program execution to allow us to change the current watched variables according to the schedule? In this section we ignore question 1 for now.

At the initial level, we of course have the DBP registers themselves. We assume that on each trigger of our instrumentation, we can know the current statement in the program. Also note that DBP triggers are not necessarily lightweight. Our current mechanism (and the only known capability) places triggers in a parent process (similar to a debugger); thus a trigger causes a heavyweight context switch to another process.[1]

If the data breakpoints themselves are not sufficient, then what other instrumentation do we need? A first step would be function entry and exit, with a more generalized notion being basic block entry and exit. Having access to the scope entry and exit of variables that are being watched is of obvious benefit. It may also be the case that schedule changes might need to be done within a scope but still at the beginning or end of some intermediate basic block.

The final, most detailed level, would be the ability to instrument arbitrary points in the program, between individual statements and even at the expression level. This would allow schedule changes at any point in the program. At this level our task seems essentially equivalent to register allocation. It is important to remember, however, that modifying the DBP registers is likely to be a heavyweight operation (e.g., involving a context switch), and thus its frequency needs to be minimized. Again, experimental evidence will be needed to decide how often this level of instrumentation will be needed.

It is important to remember that the DBP registers watch for accesses to a memory location. They do *not* attempt to correspond data register accesses to memory location accesses. Thus, as long as variable values are in a register and being used in a register, the variable accesses are invisible to the DBP mechanism. The full implications of this are not yet absorbed by the authors, but it certainly implies that DBP registers do not necessarily provide full trapping of all variable accesses, especially in optimized programs and on architectures with a large number of registers.

3. Watching multiple variables

The first scenario that we consider is the simple case of watching more variables than there are breakpoint registers. In this section we are assuming that the program does not use pointers.

In considering how to watch the variables, the first thing we need to look at is the lifetimes of the variables. If the number of overlapping variable lifetimes is less than or equal to the number of breakpoint registers, then watching them is fairly straightforward. It still requires, however, a schedule for which variables are watched when, and the selection of triggers on which to change the schedule. Because DBP registers watch addresses, it would be incorrect to watch a local variable's address while out of scope for that variable. Thus, even without overlapping lifetimes outnumbering our DBP

[1]Initial measurements over a simple program where a trigger occurred every 10th iteration through a counting loop resulted in a slowdown of about 50—the original unwatched process took 12.41 seconds, while the watched process (and the monitor) took 116.1 user seconds and 502.51 system seconds. While much better than the 10,400 slowdown of the debugger, this is quite high for hardware-assisted breakpoints, and the numbers show that most of the overhead is in the kernel.

registers, scheduling triggers may still need to resort to instrumentation at the scope entry/exit level. It should never need any finer level of instrumentation for this case.

Next is the case where the number of overlapping lifetimes is greater than the number of DBP registers. Here there can be no guaranteed minimal instrumentation level. Yet there still can be hope for the higher levels. If static analysis can determine a set of variable watches that must trigger and that can indicate a point to change one or more DBP registers to watch other variables, then even this case might be handled by the highest level of instrumentation.

We think that the best way to evaluate this is to analyze some benchmark programs over a variety of watch sets and determine what is needed.

4. Introducing pointers

Pointers, as always, are the bane of analyzers. Moreover, pointers are probably the exact cause of many of the problems we might want to find by watching variables. A programmer wants to know exactly when and where a variable is first getting clobbered.

Many studies have been done on points-to analyses, and it is often the case that pointers have very small points-to sets (e.g., [2]). This is encouraging in that it provides hope that the potential program sites for reading and writing watched variables does not greatly increase with pointers in the code. However, with even one pointer having a large or all-variables points-to set (including our watched variables, one of which it may be clobbering!) and a watch set greater than the number of DBP registers, we are immediately in trouble in how to set our DBP registers.

A potential solution is, in these cases, to watch the pointer rather than the variables in our watch set. This allows us, with one DBP register while in the scope of that pointer, to be able to determine if the accesses of the pointer will read or write any of the variables. We can then "virtually" trigger the instrumentation on the variables themselves. This idea can be applied not just at the scoping level of troublesome pointers, but at the statement level. Thus, essentially, they become additional variables to watch, with an even higher priority than our regular variables.

A downside to this approach is that the slowdown of the program might greatly increase due to the interruptions caused by pointer accesses. For non-safe pointers we would potentially need to trap every read of the pointer because it could cause a write to one of our variables. We should be able to eliminate through static analysis the program locations where the pointers are only being accessed to read the data, but this may be infrequent or not beneficial in average programs. We could also skip pointer reads when we know the current value of the pointer is not one of our watched variables, and only trap on the next pointer write (assuming no pointer arithmetic).

5. Static analyses

Since we are considering the problem of watching data accesses, it seems natural that dataflow analyses are the types of analyses that would most directly inform our dynamic analysis stage.

With def-use information and points-to sets from a pointer analysis, we would know where in the CFG does each watched variable need to be watched. If variables are being watched only for writes, then blocks with variable definitions matter. If we are watching reads and writes, then both defs and uses matter.

We assume that we can have a statement or expression-level CFG if needed, rather than simply a basic-block CFG. This would allow us to ensure that there is no CFG node does not indicate more watched variables than are physically possible.

With the above information, the essential problem is that of creating an efficient schedule. By efficient we mean one that is cost-minimally updated. This may not be the same as one with the minimal number of updates. For example, since DBP triggers already interrupt the process when they occur, it would probably be cheaper to update the schedule from within DBP triggers even if we need extra or more updates than with a scheme that needed to add special traps (causing new context switches) to achieve an absolute minimum number of updates.

Attacking this problem might introduce some new analyses that have heretofore not been considered. For example, if we are using write-only DBP triggers, then if a watched variable write is dominated by a set of writes on currently watched variables, then that set can potentially serve as schedule update points to bring in the new variable needing watched. In other words, we need to find the def(X)-def(Y) chains, where a definition of watched variable X is live at the definition of watched variable Y. Finding dominance relations over these chains would give us points for potential schedule changes in the DBP triggers themselves.

Similar inter-variable dataflow analysis would be needed for variables being watched for read and write accesses. While it sounds daunting at first, this type of analysis would only need performed over watched variables (and some pointers), not all variables, and may be an effective (or necessary) way of finding schedule updates based on the DBP triggers themselves.

6. Statistical tracing

For some programs and set of desired variables to watch, it might be the case that 100% coverage of all variable accesses is simply too prohibitive in cost to achieve. This might be because the program has many ill-defined pointers that need to be watched constantly and thus cause many program interrupts, or because the number of watched variables and their interaction is such that DBP register schedule changes need to be made so often that it results in too much instrumentation overhead.

Thus, we may wish to attempt to catch *most* variable accesses, but with much less instrumentation. Rather than consider the whole space a continuum over which to make this tradeoff, for now we simplify the problem to the following question. With DBP triggers and function entry/exit triggers, can we schedule the DBP registers to catch a high percentage of watched variable accesses?

We feel that an empirical investigation into this will be the only way to really answer the question, given the range of possible programs and specifications of watched variables that can be involved. While there will likely be no guaranteable achieved coverage, perhaps a static analysis phase could optimize the schedule and warn about likely code areas where large numbers of variable accesses may be missed.

A somewhat tangential but related idea is that of saving the previous value of each watched variable at each DBP trigger. This would only require a doubling (plus some overhead) of the watched variable space, and would provide a safeguard mechanism for potentially noting missed writes on watched variables. If the current value at a DBP trigger is the same as the previous, we cannot say for certain that there was no write in between (it might have written the same value), but if it is different we have definitely detected an intervening write that was missed by our DBP triggers.

7. Related work

Wahbe et al. [3] present the closest related work, in which they attack the problem of data breakpoints. Their motivation is the same, and they give an example of a slowdown of 85,000 when data breakpoints are used in a debugger. Interestingly enough, they mention the existence of data breakpoint registers, but do not use them in their work. They dismiss them because of their limited numbers (the Intel i386+, at four, seem to have the most). Rather, they take a code-patching approach, and they do employ some static analysis steps to reduce the amount of instrumentation.

Ball and Larus discuss the optimization of program tracing in [1], but their work is focused on control-flow tracing, and optimal placement of instrumentation to capture enough information to reconstruct the original control flow.

8. Conclusion

Data breakpoint registers, although few, offer hardware support for dynamic analyses that need to observe data accesses. In trying to create efficient instrumentation for dynamic analyses, we should use, as best we can, every resource that is available. To this end, we presented ideas for how the data breakpoint registers might be used and managed to watch a large number of variables.

Our ideas center around performing some static analysis in order to determine a schedule of DBP allocation that will cover the variable accesses we are interested in. Some harder issues that we have not yet thought about are multithreaded programs with global variables, shared memory pages between processes, and other mechanisms that step outside of the bounds of single-thread access to data.

On the practical side, it is interesting to note that the only implementation support for using data breakpoint registers is highly inefficient, forcing a context switch to a monitoring (parent) process. While this may be natural for user-controlled debuggers to use, automatic runtime monitors would benefit from new, efficient support for these hardware resources.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-0306457, EIA-9810732, and EIA-0220590. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] T. Ball and J. Larus. Optimally Profiling and Tracing Programs. 16(4):1319–1360, July 1994.

[2] M. Hind and A. Pioli. Which Pointer Analysis Should I Use. In *Proc. 2000 International Symposium on Software Testing and Analysis*, Aug. 2000.

[3] R. WAhbe, S. Lucco, and S. Graham. Practical Data Breakpoints: Design and Implementation. In *Proc. 1993 Conference on Programming Language Design and Implementation*, pages 1–12, June 1993.