IMPLEMENTATION AND EVALUATION OF DATA BREAKPOINT SCHEMES IN AN INTERACTIVE DEBUGGER

by

Paul E. Roberts

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

December 1996

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Paul E. Roberts

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: John B. Carter

Robert R. Kessler

Gary E. Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Paul E. Roberts in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

John B. Carter
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

Interactive debuggers allow users to specify conditions upon which the debugger stops the program being debugged so that the user can examine the state of the program. This stopping point is generally called a *breakpoint*. The most commonly used form of breakpoint, a control breakpoint, specifies the place in the program to stop. This type of breakpoint is well understood and implemented efficiently in most debuggers. *Watchpoints* are a special kind of breakpoint, a data breakpoint, that stop the program when the value of an expression changes. It has proven difficult to develop efficient watchpoint implementations and although a number of efficient mechanisms have been proposed, most debuggers still use the simplest and slowest. I have implemented variants of watchpoints in GDB, the GNU source-level debugger. This work shows that there are a number of practical implementations of watchpoints that perform much better than the common implementation. The watchpoint implementations that I evaluated outperformed GDB's simplistic implementation by factors of 20 to 50,000. I conclude that the most efficient implementations do much of the work outside of the debugger, that is, either in the operating system kernel or in the program being debugged.

For Sussy.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Jay Lepreau for his help, ideas and support for the project, and Jeff Law for his technical support. I would also like to thank John Carter for ideas on the project and his help in writing the thesis, and Bob Kessler and Gary Lindstrom for their input on the thesis. I especially want to thank my wife for her loving patience and understanding.

CHAPTER 1

INTRODUCTION

Watchpoints [22] in GDB, the GNU source-level debugger, are a special kind of breakpoint, a data breakpoint. In general, breakpoints are the mechanism used to specify under which conditions a debugger should stop the program being debugged and perform some action, such as allowing the user to examine the state of the program. The most common form of breakpoint is a control breakpoint, which specifies the place in the program where the user wants it to stop. Control breakpoints are well understood and implemented efficiently in GDB. Watchpoints, on the other hand, are special breakpoints that stop the program when the value of an expression changes. For example if the program had a variable called x, a watchpoint on the expression x would stop the program anytime the value associated with the variable x changed. Watchpoints are not implemented efficiently in most debuggers including GDB. This thesis shows that dramatic improvements in the performance of watchpoints can be made in a variety of ways. This thesis also compares these solutions for efficient watchpoints with each other.

Having an efficient implementation of watchpoints would help programmers immensely in debugging programs [24, 17, 11]. Without watchpoints it is difficult to find problems with pointer uses that result in inadvertent modification of unrelated data. For example, if there is a bug in the program that incorrectly manipulates a pointer so that it points to memory associated with a different data structure and then that memory is written, it is often the case that the problem is not noticed until the data structure that was inadvertently modified is referenced. This could be a long time after the erroneous modification was made, long after all clues as to when it was modified have been lost. With watchpoints this problem is solved by setting a watchpoint on the data structure before the inadvertent modification. The debugger then stops the program when the data are modified. With knowledge of the time that the data are modified, the user is often much closer to determining the root of the problem. Without watchpoints it could be a long and tedious process to find when the inadvertent modification occurred.

Also, having watchpoints could help a programmer understand what is happening in a large program with which he is not very familiar. If a function is using global data that were previously initialized, it is often useful to know where the data came from to understand what the function and the program are doing and to determine if they are actually doing what they are supposed to do. Clearly the biggest need for watchpoints is for monitoring data in large programs. Unfortunately the implementation of watchpoints in GDB is so slow that they can be used effectively only in very small programs or very small parts of large programs. Thus the major benefit of watchpoints is eliminated.

Watchpoints in GDB are slow because they are implemented using single stepping. GDB is not the only debugger to use single stepping. Most UNIX debuggers and many others use this approach, for example SUN's dbx [19], HP's xdb [10], and Hitachi's sdb [20]. Single stepping consists of stopping the program after each instruction, returning control to the debugger, and evaluating the expressions associated with any watchpoints to see if they have changed. If not, the debugger returns control to the program for one more instruction, and so on. This implementation, although very simple and portable, is extremely slow, requiring one trap and several crossings of protection domains for each instruction.

In analyzing the speed of watchpoints, there are three operations that we must consider: inserting the watchpoint, removing the watchpoint, and checking if the watchpoint condition is met. At the very least, inserting and removing the watchpoint needs to occur each time the user specifies that a watchpoint is to be inserted or removed. At most, each active watchpoint needs to be inserted each time the debugger starts or continues execution of the program and removed each time it stops execution. In GDB, breakpoints and watchpoints are removed and inserted each time the execution stops and starts. Strictly speaking, it is not necessary to remove and reinsert the watchpoint each time the program stops and restarts but there are times that this is necessary. To avoid the complications associated with determining when it is and is not necessary, they are always removed and reinserted. Since GDB, which is an interactive debugger, is being considered, the overhead of starting and stopping the program (i.e., inserting and removing watchpoints) is relatively unimportant as they each happen only once before the debugger must wait for the user. So, as long as they are less than a tenth of a second, the user probably will not notice any difference. On the other hand, checking the watchpoint condition may need to be done any number of times while the program is running. Thus, the key to fast watchpoints is to reduce as much as possible the time required to evaluate the watchpoint condition.

There are basically two ways to speed up watchpoints. One is to reduce the number of times the check is made to see if the expression has changed. The other is to make the check faster. There are three ways that have been proposed to reduce the number of checks. The first is to check only write instructions instead of all instructions [24, 17]; the second is to check only the write instructions that write to the same page as the watched memory [24, 17]; the third is to checkpoint [17], that is, periodically stop and check the condition with the ability to roll back to a previous state if the stopping condition has been met. There are many ways proposed to make the check faster, from moving the checks outside of the debugger and into the kernel or the debuggee itself [24, 17], to using special hardware that may be available [18, 24, 17], to just using more efficient data structures and algorithms to do the checks [25], and various combinations of the above.

Ideally watchpoints would be implemented so there is no noticeable slow down in runtime, without any changes in the semantics of watchpoints or the program being debugged, without the need for operating system support or the existence of other tools, and without requiring too much effort by the developer or the user. However, this is not an ideal world. Watchpoints would be useful if the slow down was not too great and as few of the other changes were required as possible. Realizing that great improvements in speed may be obtained by compromising some of these ideals, the success of an implementation will be judged not only on the speed but also on how bearable the compromises to the other ideals are.

I have studied a number of proposed solutions and will discuss in more detail the merits and drawbacks of each proposed solution in the following chapter of this thesis. The proposed solutions are trap patching, code patching, checkpointing, virtual memory, HP break bit, kernel checking, patch on trap, and monitor registers. I have implemented a few of the most promising solutions: code patching, virtual memory, HP break bit, and kernel checking. I will compare them in terms of speed, portability, and ease of implementation. I will also discuss why I chose not to implement the other proposed solutions. Although all solutions appear to be much faster than the simple method GDB now uses for watchpoints, results show that the fastest solutions are those which do the checking outside of GDB—either in the kernel (kernel checking) or in the debuggee itself (code patching). Results indicate that most of the time the fastest solutions are less than two times slower while watching an expression than it is for normal execution without watching an expression. Watching a variable in the original implementation of GDB is about four orders of magnitude slower.

CHAPTER 2

BACKGROUND

2.1 Possible Solutions

This section discusses in general the various possible implementations that have been proposed. This discussion will refer to the C code sample and its corresponding assembly language translation found in the appendix.

2.1.1 Trap Patching

The most obvious way to reduce the number of times the check for the watchpoint condition is made is to realize that only instructions that write to memory can change the value of the expression. Thus it is only necessary to check the watchpoint condition after write instructions. Trap patching involves replacing all write instructions with a trap instruction. When the trap instruction executes, control is given to the debugger which determines whether the user specified conditions are met and if it should trigger the debugging action or emulate the write instruction and continue [24].

In the sample program there are 10 instructions that write to memory (nine *stw* instructions and one *stwm* instruction). In trap patching all of these write instructions are replaced by trap instructions. This could be done at runtime by the debugger or the program could be modified by the compiler, linker, assembler or a separate program before debugging to facilitate the debugger. However, as we shall see, other methods are clearly faster than trap patching and appear to be easier to implement; therefore there is no real need to implement trap patching for the purpose of this study.

A couple of papers describe trap patching as a possible solution for fast watchpoints [24, 17] and its performance is analyzed in a simulation study [24], however these papers do not claim that trap patching is a good solution for watchpoints nor do they cite any implementations of this approach.

2.1.2 Code Patching

Like trap patching, code patching checks every single write. Instead of trapping and returning control to the debugger, however, the check is done by the debuggee itself. This requires adding code to the executable to be debugged that checks the target of every write instruction [24]. To implement code patching, both the debugger and debuggee must be modified. The debuggee must have its code patched. The patching could be accomplished by the compiler, the assembler, the linker, the debugger, or a completely separate program.

To patch the program sample, code is added after every write instruction that checks the target of the write against a list of watched addresses. This check could be done inline or a subroutine could be called to do the check. Alternatively, the write could be replaced by a jump instruction that jumps to code that does the write and then checks the watchpoint condition and returns to the next instruction.

There has been considerable research into uses of code patching. Code patching was first suggested as a means to implement fast control breakpoints in a non-interactive environment [18]. On a SPARC workstation, code patching improved breakpoint performance by a factor of 1000 compared to standard Unix debuggers.

Later, code patching was advocated as a good solution for watchpoints in a simulation study comparing four watchpoint schemes: monitor registers (special registers dedicated to facilitate watchpoints), simple virtual memory, trap patching, and code patching [24]. This study was motivated to justify supporting code patching in the Ensemble software development environment being built at the University of California, Berkeley. The results of this simulation indicated that on average the monitor register scheme performed best, the code patching scheme slightly outperformed the virtual memory scheme, and trap patching performed worst. However, the worst case results were quite different, with code patching having by far the best worst case and virtual memory having a very bad worst case. For the most part, the analytical models used in this study are fair except that they completely ignore the costs of communication and of context switches between the debugger and the debuggee. In my implementations these costs particularly worsen the performance of code patching when inserting and deleting watchpoints.

Code patching has since been implemented at the University of California, Berkeley and further research has been done on possible improvements to the patched code speed [25]. The researchers concluded that the data structure used to store the watched addresses is of particular importance. They found that a bitmap was better than a hash table and a segmented bit map was even better. Also, reserving registers to avoid spilling and the recalculation of address constants improved performance. Inlining the check did not make a difference and segment caching and trying to eliminate unnecessary write checks by using data flow analysis were not worth the effort.

As a result of the code patching research being done at the University of California, Berkeley, code patching is beginning to be implemented in some debugging environments to facilitate watchpoints. For example, Kubota Pacific Computer, Inc. has modified their debugger dbg [11].

Independent of the research at the University of California, Berkeley, a company called Pure Software Inc. has also done extensive research into the uses of code patching including its use for detection of memory leaks and access errors. [14].

Still others have modified GDB to provide kernel debugging support for Mach micro-kernels on the Motorola 88100/88200 [2]. Part of this work included a dynamic code generator to do code patching for conditional control breakpoints. Dynamic code patching improved the performance of some conditional control breakpoints by a factor of 1000. However, this implementation works only for debugging the kernel and not user programs. Also, watchpoints are a bit more difficult to implement than conditional breakpoints because one needs a way to identify the write instructions dynamically. Finally, the Los Alamos debugger ldb uses code patching to do fast conditional control breakpoints [4]

2.1.3 Checkpointing

Another way to reduce the number of checks made is checkpointing [17]. This scheme is interesting, but has a major flaw in addition to the normal difficulties of doing checkpointing correctly [26] so I did not implement it. The idea behind checkpointing is the debugger periodically stops the program, saves the state, and checks the condition for stopping. If the condition for stopping is true, then the debugger restarts the program from the point of the previous checkpoint, only this time using a smaller interval to checkpoint; otherwise it just continues executing the program after saving the state. The debugger repeats this process until it defaults to single stepping and the place where the condition for stopping is found. At first this may seem like a very good solution, but it has a major problem. It is possible to miss the point that the condition changes if the condition for stopping becomes true but then becomes false again before the next checkpoint.

In the program sample in the appendix, if the expression $p \rightarrow q$ were to be watched from the beginning of the loop, the debugger would save all of the state and remember that the value of the expression is 0x0. It would then execute some number of instructions (e.g., 1000) and find that the value has changed. The debugger would then restart the program from the previous spot after restoring the state, this time stopping after a smaller number of instructions (e.g., 100). If the program just happened to stop after incrementing p but before the next initialization of $p \rightarrow q$ then the debugger would see that the value of the expression is still 0x0. It would assume that the value has not changed, even though it has changed many times, and continue from the new spot after saving all of the state. Although checkpointing is described as a possible solution for watchpoints [17], it has not been implemented nor has its performance been analyzed.

2.1.4 Virtual Memory

The number of checks can be further reduced by checking only those writes that write to pages that contain data involved in a watched expression. In other words, use the virtual memory system to protect the pages containing the data involved in such expressions. When a write to a protected page occurs, the debugger handles the fault. The debugger unprotects the page, allows the write to occur, and reprotects the page. It then checks to see if the specified conditions are met and triggers the debugging action or continues [24].

In the program sample, the variables a and i happen to be on the same page while p and $result$ are on the next page. The variable k is on a third page associated with the stack and the malloc'd memory comes from another page associated with the heap. See Figure 2.1. If the expression p were to be watched, only those writes to p, $result$, and perhaps memory that we are not aware of would cause a fault, but not writes to a, i, or k. However if $p \rightarrow q$ were to be watched while p points into a, then both pages would be protected and all writes to p, $result$, i and a would cause a page fault.

VAX DEBUG uses the virtual memory system to do watchpoints [1] but performance information is not available for this implementation. I have been told of a

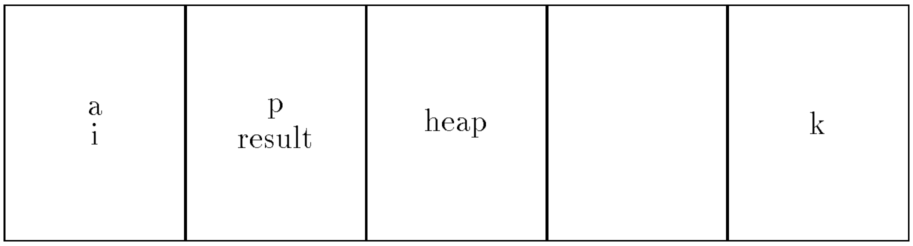

Figure 2.1. Illustration of memory layout of sample program

debugger written for the Apple Macintosh that uses its MMU to do watchpoints [6], but have been unable to confirm this. I am unaware of any other implementations that use the virtual memory system to do watchpoints. Only the simulation study mentioned earlier provides performance data for this type of implementation [24].

2.1.5 HP break bit

A down side to using virtual memory to implement watchpoints is that every stop and check requires a page fault, two system calls to unprotect and then protect the page and a trap to single step the write instruction—a heavy overhead. On the HP PA, the virtual memory scheme can be improved [8] because its virtual memory system has a special bit called the B-bit in its TLB. The B-bit when used in conjunction with the X-bit in the PSW is designed specifically to tell a debugger that a particular page is about to be written without requiring the debugger to unprotect the page, single step the write operation, and then re-protect the page. The Connection Machine CM-5 has even better debugging support, providing similar break bits associated with each word (32 bytes) of memory [21].

Surprisingly, HP debuggers do not take advantage of the break bit feature of the PA [10]. In fact, the HPUX operating system does not even provide the mechanism to set the bit [9]. However, after hearing of my work, plans to provide such a mechanism are being made.

2.1.6 Kernel Checking

The performance of the virtual memory based schemes and even the trap patching scheme could be improved by having the operating system kernel keep track of what memory locations are being watched [24]. The fault handler then could compare the address against its list of addresses involved in a watchpoint and return control to the debugger only after the program writes to a watched address instead of every time a write to the page occurs. This design saves two context switches, from kernel to debugger and from debugger back to kernel, on "false hits."

In papers on fast data breakpoints, kernel checking is mentioned only in passing as a way of improving performance of virtual memory based implementations [24, 17]. No analysis of how much improvement kernel checking provides is attempted.

2.1.7 Patch on Trap

Patch on trap is an attempt to take the best ideas from both code patching and virtual memory schemes [17]. I did not implement patch on trap but it is an interesting proposal and deserves discussion of its merits. Patch on trap makes the assumption that if a single write instruction is repeated several times, then it is accessing the same page several times. With this assumption in mind, patch on trap combines the two ideas using the virtual memory to filter the write instructions, and then patching those write instructions in case they are ever executed again.

From the code sample, if GDB were to watch *result*, GDB would protect *result*'s page as in the normal VM scheme. However, when an instruction writes to that page, for example the instruction that increments p, GDB patches the faulting instruction so that the code jumps to a routine that performs the write and then checks the address of the write. In the process of patching the instruction, GDB maps the physical page to a second virtual page that is not protected and uses the address corresponding to the new page so that particular instruction does not cause additional page faults.

It has been proposed that the dynamic code generator, used to improve conditional control breakpoints for the kernel debugger that was described above also be used as the basis for the trap on patch scheme [17]. However, the designer of this system has told me that he has not yet implemented it, nor does he have time to do it in the immediate future.

2.1.8 Monitor Registers

Some processors, including the Intel i386 and the MIPS R4000 [12, 15], have special registers dedicated to monitoring writes. These registers hold address ranges for the memory they are monitoring and, when a write occurs to a monitored region, a trap occurs. Although this hardware solution is fast, it is limited by the number of registers dedicated to monitoring writes. I am not implementing this solution because the machine I am using does not have monitor registers.

Since Intel 386 based PC's have monitor registers, most of the better code development packages for PC's from commercial vendors take advantage of them to do watchpoints. I could only find one Unix debugger, the dynix/ptx version of USL's debug, that uses monitor registers, but it resorts to single stepping when it runs out of monitor registers [13, 23]. Another debugger, PER, which runs on the MTS operating system, also uses monitor registers [5]. GDB's developers plan to modify GDB to use them on those architectures that have them [22].

2.1.9 Other Solutions

Implementing watchpoints in interpreted code development environments is not difficult. The interpreter can be modified to do something special when it encounters an instruction that writes memory. Likewise, machine simulators can be altered to check writes and a debugger could then execute the code using the simulator instead of the actual machine.

2.2 Expected Performance

Before presenting detailed performance results, I first present and informal comparison of the expected results of the various implementations. The comparison is based on how many times a given implementation is expected to check the watchpoint condition and how much time is expected to be used to do the check in the implementation being considered.

A trap patching implementation will perform better than a single stepping implementation because the program stops and returns control to the debugger only after write instructions and not after every instruction. However a trap patching implementation will perform worse than a code patching one. They both perform the same number of checks, but the overhead of the check of a trap patching implementation is much greater than that of a code patching implementation, since in a trap patching implementation control is returned to the debugger whereas in a code patching implementation the check is done by the debuggee itself. Also a trap patching implementation would almost always perform more poorly than a virtual memory or HP PA break bit implementation. The overhead of the trap patching check would be about the same as the HP break bit check and about half of the overhead of a virtual memory check, but the trap patching implementation would most certainly have to perform more checks. It could not perform better than the HP break bit implementation and could only perform better than the virtual memory implementation if the virtual memory implementation checks more than half of the writes.

A code patching implementation will perform better than most other implementations. Though it checks every write instruction, the check is very fast because it is done by the debuggee itself. A code patching implementation could perform worse than a virtual memory or HP PA break bit implementation if the virtual memory or HP PA break bit implementation does very few checks; but if they do very many checks, then a code patching implementation would be much faster. One problem with a code patching implementation is that the performance is worsened even when not watching an expression because the added code is executed whether an expression is being watched or not.

It is difficult to evaluate how a checkpointing implementation will perform. It will require considerable tuning to get optimal performance, and the tuning for optimal performance will vary from program to program, and even from expression to expression in the same program. Optimal performance cannot be obtained with a general algorithm for determining when to stop. The performance would be determined by the choice of algorithm used for determining when to stop and the behavior of the expression being watched. Since I am not implementing this solution because of the flaw in it, I will not investigate different algorithms for determining when to stop nor consider its performance further.

The performance of a virtual memory based implementation is dependent upon the program being debugged and the expression being watched. If the memory associated with the expression being watched resides on a page that is rarely written, then the performance of a virtual memory implementation will be very good. On the other hand, if the protected page is written millions of times, then the performance will be very poor. As a general rule of thumb, very large programs with data spread out over many pages should generally perform well, as should programs that do not write to memory much; whereas, small write-intensive programs will not perform well.

The performance of an HP PA break bit based implementation will have the same characteristics as a virtual memory based implementation except that it should perform at least two times faster given that it does not have to unprotect the page, single step, and then reprotect the page.

Adding kernel checking to the virtual memory and HP PA break bit based implementation should improve performance of those implementations significantly. These implementations will have the same performance trends but with much lower constants as the implementations without kernel checking. Thus they will require many more writes to a protected page before the performance degrades.

A trap on patch implementation will perform better than a code patching implementation only when a small portion of the write instructions are patched. Still, the performance characteristics of the two implementations should be comparable. The main reason one would forgo the easier, more portable implementation of code patching in favor of the more difficult, less portable trap on patch implementation would not be for performance reasons, but rather to avoid having to modify the debuggee before it can be debugged.

Likewise, a trap on patch implementation will perform better than a virtual memory based implementation only when the patched instruction is executed many times. It is difficult and expensive to patch instructions dynamically, so patching an instruction that only executes a few times will result in a net loss in performance. However, trap on patch does avoid the worst case problems associated with a virtual memory based implementation, so if the worst case is common or is unacceptable to a particular user, then a trap on patch implementation may be a viable solution.

A monitor register based implementation will be the overall fastest solution except that such an implementation is limited by the number of monitor registers that the architecture has. When that limit is exceeded, some other strategy is required and the performance of the implementation suffers.

Using an interpreter to implement watchpoints would not be efficient because interpreted code runs slower than compiled code, but it would be faster than single stepping. For example, interpreted ANSI C runs approximately 200 times slower than compiled code [16]. Likewise, machine simulators tend to run 5 to 50 times slower than native hardware [3, 7].

CHAPTER 3

IMPLEMENTATION DETAILS

Many of the techniques proposed to provide efficient watchpoints rely on a service, either abstract or real, that informs the debugger when a particular address in memory is written. To accomplish this, the debugger must be able to determine what addresses in memory need to be watched when given the expression to be watched. This is not as simple as just watching a single location or block in memory, which is what most papers on fast data breakpoints assume, because in GDB one can watch an arbitrarily complex expression. GDB is not alone in allowing one to watch complex expressions. Many other debuggers can watch complex expressions, although some limit watchpoints to memory regions and others limit them to variables or have other limitations to facilitate implementation.

From the sample program in the appendix, $p \to q \to r$ and $i + k$ are valid expressions one can watch. Both of these expressions have more than one memory location that when written could change the value of the expression. In GDB there is a simple mechanism to determine all memory locations that are used to determine the value of an expression. After an expression has been evaluated, all of the information used to determine the value of the expression is saved in a data structure implemented as a linked list. By scanning the list for all memory locations, all locations that could affect the value of the expression can be found. Unfortunately, this method also often includes memory locations that never affect the value of the expression. This is most evident when watching elements of arrays or fields of structs— in these cases the whole data structure is watched. Since correctness is more important than speed, this simple mechanism for determining which memory locations to watch is used. If this simple mechanism introduces too much overhead, a more sophisticated method could be employed.

3.1 Trap Patching

Although I did not implement trap patching, for completeness I discuss how it could be implemented. To implement trap patching at runtime in GDB, there needs to be a mechanism to determine where all the write instructions are in the program and place traps there. This requires scanning the whole program an instruction at a time and replacing write instructions with traps. This is made easier with the advent of RISC processors with simple load–store semantics. GDB also needs to remember what each of these instructions is. This scanning only needs to be done once if GDB remembers the addresses of all the write instructions. Nevertheless, inserting and removing watchpoints remains an expensive operation because each insertion and deletion requires at least one system call for each write instruction to replace the instruction. Also the ability to correctly emulate write instructions is needed, which could be difficult to do correctly. Furthermore. having GDB do the trap patching at runtime would not be very portable.

Alternatively, the program could be altered by the compiler, assembler, linker or other program before debugging the program. This would make trap patching much easier to implement, but would also introduce the undesirable consequences of modifying the program. Furthermore, if one is to modify the program, he might as well patch the check right into the program, in other words, do code patching.

3.2 Code Patching

For a debugger to use the code patching technique of implementing watchpoints, there must be a tool to patch the code. As far as efficiency of watchpoints is concerned, it would make little difference if the compiler, assembler, linker or a completely separate program is used to do the patching. This is because essentially the same instructions need to be executed regardless of how the program is patched. Doing the patching dynamically in the debugger would most likely be slower because the time to do the patching would be part of running the program. Furthermore, doing the patching dynamically is much more difficult to implement and makes the implementation less portable.

In my evaluation, the compiler (gcc) was modified to do the code patching because this approach was the easiest to implement. This design, however, has the drawbacks of requiring the executables to be recompiled before fast watchpoints can be utilized and of requiring patched libraries if catching writes in library routines is necessary. Some may argue that it would be better to have a separate program do the patching, which would make it easier for the user to patch existing code. This is a valid argument, but is irrelevant to the performance of code patching.

In my implementation, after each write instruction the compiler inserts a procedure call to the routine that monitors the watchpoints, passing the address of the write to the procedure via a register. The procedure returns either zero or one. After the return, a trap instruction is conditionally executed depending upon the return value. Some may argue that it would be better to replace the writes with jumps to the patched code that does the write and calls the routine that monitors the watchpoints instead of inserting the code after the write. This separates the patched code from the original code which could eliminate some heisenbugs[1] caused

[1]Bugs that go away when one looks for them.

by modifying the program. My code patching design decisions are based on getting a quick and painless implementation working correctly that generates valid timings without worrying about what is the best implementation from a user's point of view.

The patching is accomplished by inserting a procedure call to *check_watch*() after each write instruction and then trapping if *check_watch*() returns true; *check_watch*(), along with *insert_watch*() and *remove_watch*(), are functions added to the libgcc2 library. These functions read and update a segmented bitmap that keeps track of watchpoints. From the program sample, an instruction such as:

```
        stw %r20,0(0,%r19)       ;write a word to memory
```

would be patched to become:

```
        stw %r20,0(0,%r19)       ;write a word to memory
        copy %r19,%r26           ;put address of memory location in reserved
        .CALL ARGW0=GR           ;  register to be used by procedure call
        bl _check_watch,%r2      ;call _check_watch procedure
        nop
        comiclr,<> 0,%r28,0      ;compare result to zero
        bl,n L$0005,0            ;goto label if result was zero
        break 4,8                ;trap
L$0005:
```

For GDB to take advantage of the patched code, some modifications to GDB are required. First, GDB must know if it is dealing with a patched program or not—if not then GDB handles watchpoints the old way. GDB makes this determination by searching its symbol table for one of the three library functions that are added to the patched code.

Second, GDB must be able to insert and remove watchpoints correctly. To do this, GDB uses its ability to dynamically call code in the debuggee. This ability to call code dynamically allows the user to execute any function in the program while the program is stopped somewhere during its execution. From a software engineering point of view this is the ideal way to insert and remove watchpoints: the debugger simply calls the appropriate functions in the inferior process. However, using the dynamic function calling feature did present some problems. First, the code that calls the functions dynamically proved to be very fragile and it took considerable effort to get it to insert and remove watchpoints without breaking. Second, the dynamic function calling code is extremely slow due to its excessive use of *ptrace*. Although in most cases this poor performance is not a big concern, it turned out there were some cases in which it severely hampered the performance of the code patching implementation.

Third, GDB must realize when it has stopped because it hit a trap instruction in the patched code so that it can properly ignore that instruction when it tries to start again or it remains stuck there forever. This is not too difficult, a few lines of code in the appropriate place that checks the instruction and increments the PC if a trap instruction is found accomplishes the task.

The code patching implementation is very portable in that it does not rely on any hardware or operating system support. All of the modifications are made to user code and there appears to be no reason why it could not be implemented on any architecture. However it is more invasive and demands more work to implement than many of the other solutions. It also requires that the user be aware of the implementation; the user must have his or her code patched or no benefit is gained from the improved implementation. If code patching is done correctly, it should not change the behavior of the debuggee, however with the code expansion and addition of data structures and functionality to check for watchpoints the possibility exists that unforseen changes to the program behavior could arise. Furthermore, there is no guarantee that a program with data corruption problems will not corrupt the data used to check the watchpoints. If this occurs, the checking may no longer work correctly.

3.3 Checkpointing

The checkpointing scheme requires modifying GDB to set up a special temporary conditional breakpoint that either automatically saves all the information necessary to start up from that point again or starts up at the previous checkpoint using a smaller interval before it stops again. Also a mechanism to determine where to set up this special breakpoint is required. The first part should not be too difficult to implement although it would take a considerable amount of work to save and restore all of the state and may require considerable memory resources. The latter part would be simple if an instruction counter or a timer existed, otherwise it is unclear how it could be done. Since this scheme has a major flaw, it is an interesting but useless solution and is not implemented.

3.4 Virtual Memory

To implement the virtual memory based solution, GDB must be able to insert and delete the watchpoints, in other words, protect and unprotect the pages corresponding to the data involved. To do this, I added a *ptrace* call in the kernel that calls an mprotect-like function. Also, I added some complexity to the wait-for-inferior function in GDB to determine if the debuggee is stopped because of a watchpoint. This added code is needed so that GDB can correctly step over the instruction that is causing the fault. Determining if the debuggee was stopped because of a watchpoint requires functions that when given an address can tell not only if that address is being watched but also if there is an address on the same page that is being watched. This mechanism is necessary since the faulting signal is SIGBUS, which may well not have been caused by a watchpoint.

This implementation is easily portable to all machines with operating systems that allow a debugger to protect pages in an inferior process. Unfortunately few systems allow this, but it can be easily added to an operating system by its developers.

3.5 HP PA Break Bit

To take advantage of the break bit, GDB must be able to set and reset the break bit. Since the kernel did not provide a mechanism to set and reset the break bit when I began this research, I modified the kernel to accomplish this.[2] Again, *ptrace* is extended to call an mprotect-like function that sets the bit. Also, I added code to the kernel to handle faults caused by the break bit. Although it is quite simple for GDB to determine if the fault is due to the break bit, to improve performance GDB is enhanced so that it can tell if a given address is being watched. Normally, GDB needs to check the expression after the write instruction has executed. It determines if it stops by comparing the value of the expression with the previous value that has been saved. However, the kernel informs GDB of a write to the page before the write instruction executes. GDB would normally be required to execute one instruction and then do its comparison. Code was added to GDB that calculates the address of the write and checks if any of the watchpoints are watching that address. If not, it simply continues and avoids the single step and comparison. This saves a trap on false hits.

Most architectures do not have a break bit so, of course, this implementation is not applicable to them. However, from the GDB point of view, the implementation is very similar to the virtual memory implementation, just a little cleaner. So there would be no problems in having GDB use the virtual memory system of other architectures and still take advantage of the break bit on the HP PA.

---

[2]After hearing of my work, Hewlett Packard plans to add this functionality to HPUX.

3.6 Kernel Checking

The performance of the virtual memory and HP PA break bit based implementations can be improved by checking the addresses of the write instructions that caused faults due to watchpoints in the operating system kernel instead of GDB. To implement kernel checking, a segmented bitmap that stores addresses being watched is added to the kernel along with functions to update and check the bitmap. The version based on the HP PA break bit is much simpler than the version based on virtual memory. Not only does the virtual memory scheme need to keep track of both the pages and the addresses being watched, but it also must unprotect the page, emulate the faulting instruction, and reprotect the page after every fault. In both cases, however, kernel modifications make the modifications to GDB simpler by removing the address checking previously done by GDB.

Adding kernel checking to the trap patching scheme appears to be more difficult to implement. A mechanism to inform the kernel which addresses are being watched is needed. This mechanism is not difficult to add, and it is already part of the VM based schemes. Also, unless the program to be debugged is modified before running, the kernel not only needs to emulate the write instructions, but it needs to keep track of all of the write instructions. Furthermore, if for some reason the kernel is unable to emulate an instruction, it does not have a fall back solution like the VM based scheme does. Moreover, the VM based schemes are clearly better than the trap patching scheme, so it was deemed to be a waste of effort to implement trap patching.

As far as portability is concerned, adding kernel checking to an operating system is not difficult for those who have access to and can change the operating system source. Of course, without the operating system support, this approach is not applicable.

3.7 Patch on Trap

Patch on trap is by far the most complicated scheme. It is not a simple combination of the code patching and virtual memory schemes. The protected pages must be double mapped to pages that are not write protected so that the patched code can write without causing a trap. Also the code patching must be done dynamically by GDB, which is more difficult and probably slower than having the compiler, assembler, or linker do it. The major obstacle to dynamic code patching is finding the space to do it. Not only is space needed for a rather large data structure, but also for code that is generated dynamically and may need to persist for an indefinite period of time. Furthermore, this scheme is not easily portable. Not only does GDB need to know how to patch code for each architecture but it seems unlikely that the special locations that it places the code for one architecture will work for other architectures. In short, watchpointing could be a nice application of a dynamic code patcher, but building a dynamic code patcher for the purpose of doing watchpoints is a waste of effort. Therefore I did not pursue or evaluate experimentally this scheme.

3.8 Monitor Registers

The hardware I am using, the HP PA, does not provide monitor registers so I did not implement a watchpoint mechanism based on them. However, the GDB manual hints there are plans to use these registers in the future for machines that provide them. In theory, basing watchpoints on monitor registers should not be much different from using virtual memory or break bits. There will probably be a system call to set the monitor registers and GDB may have to do something special to execute the faulting instruction before it makes the comparison.

However, the monitor register scheme limits the number of regions that one can watch because there are a limited number of registers. Dealing with the situation where more monitor registers are required than are available is a difficult problem. Single stepping when this condition is reached is a bad idea. One option that I believe will work well is to combine two address ranges that are fairly close to each other and watch only the combined region. Overall, the use of monitor registers is probably the fastest solution, especially for watching a few simple expressions, and therefore it should be implemented on those architectures that provide monitor registers.

CHAPTER 4

RESULTS

The most promising of the schemes discussed in Chapters 2 and 3 have been implemented on the HP PA using GDB version 4.12. The virtual memory scheme and the HP break bit scheme are implemented both with and without kernel checking. Code patching is implemented using the GNU C compiler to do the code patching.

4.1 Test Suite

To evaluate the different schemes, a test suite that consists of three C programs from the SPECmark benchmark suite, (eqntott, espresso and xlisp) and gcc version 2.5.8 as a substitute for the gcc in the SPECmark suite is used.[1] The SPECmark benchmark suite is used because it is becoming a standard for performance evaluation.

The test suite consists of GDB scripts that watch each global variable and each heap variable one at a time in the test programs. There are separate scripts for each of the global and heap variables in SPEC programs and approximately 60% of the global and heap variables in gcc. Eqntott has 53 global and three heap tests, espresso has 55 global and one heap test, xlisp has 77 global and 56 heap tests, and gcc has 812 global and 217 heap tests. For the small programs, the tests are

[1]The SPECmark version of gcc does not work on the HP PA.

run on all of the variables for completeness. Variables from the large progam were randomly chosen in order to get a representative sample.

The test suite also contains one test per program that is designed to approximate one way watchpoints could be used on local or stack variables.[2] In the test, a single memory location on the stack is watched throughout the entire duration of the running program. This test approximates watching a variable local to the main procedure but whose memory could be accessed throughout the program. This test would be similar to the way corruption of the stack would be found, which I believe to be the most legitimate use for watching a stack variable.

4.1.1 Limitations of Test Suite

Ideally, the test suite would be representative of the types of programs people debug. However, there is such a wide variety of program behaviors it may be impossible to show that a given set is representative. There are, however, well-recognized benchmark suites that have become the standard in measuring the performance of architectures, compilers, etc. The SPECmark benchmark suite is one such suite, unfortunately it may not be a good representation of the types of programs commonly debugged as they are mostly small, compute intensive programs. Furthermore, only the C programs from the SPECmark suite are used because GDB on the HP PA can not debug Fortran code.

The test suite should also use watchpoints in a representative manner. In certain cases it could be useful to watch a single element or field of an array or struct. However in the test suite used for this study, the whole data structure is watched. This is not only to limit the number of tests, but also because the simple mechanism GDB uses to determine what addresses to watch would choose to watch the whole

[2]GDB is unable to watch local variables on the PA due to an incongruity of design assumptions with the PA that GDB makes in regard to scoping.

array or struct even if the expression specifically targeted just an element or field.

There are other ways watchpoints could be used to watch local or stack variables, though many of them seem of little use. The most obvious use would be finding the place that the value of a local variable changes, however if the local variable changes in the local scope as most do, then this place is usually easy to find without watchpoints. Also, one could use watchpoints in conjunction with control breakpoints by setting a breakpoint at the beginning of a function and then watching a local variable each time the function is called; however, conditional control breakpoints seem to be better suited for this use. Still, the test from the suite is probably close to the worst case for the virtual memory based schemes, and one could expect better performance with a more limited use of watching stack variables.

None of the tests in the suite explicitly have multiple watchpoints. The heap tests implicitly watch both the pointer and the memory associated with the data. Although there may be times when multiple explicit watchpoints are convenient or even needed, most times when watchpoints are used, one is sufficient because watchpoints are most often used to find data corruption. If a write instruction corrupts data that it should not modify, it is usually a single location or contiguous memory that is modified; therefore one watchpoint is adequate. Furthermore, even if unrelated data throughout the program are being corrupted, the manifestation of these bugs will still most likely appear individually and thus will be tracked down individually. As further justification to this assumption, an informal survey of graduate students, professors, and staff members of the computer science department at the University of Utah, was conducted to determine how watchpoints are used; the results of the survey support the premise that single watchpoints are much more common than multiple watchpoints. In the test suite, multiple watchpoints are never active at one time, not only because of the reasons stated thus far, but also because in order to make a fair evaluation of having multiple watchpoints, one would have to run many more tests than time would allow, due to the many possible combinations of multiple expressions. Furthermore it would be difficult to justify why a particular set of multiple watchpoint tests is representative of how multiple watchpoints are used. However, the results of the heap tests will give some indication of how multiple watchpoints will perform, since in those tests there are two implicit watchpoints.

4.2 Legend

In the graphs presented in this chapter, the different labels represent the following: $NO\ WP$, the time it takes to run the program inside of GDB without any watchpoints set; SS, the time it takes the program to run while watching a variable using the single stepping method of implementing watchpoints; CP, the times for the code patching based implementation; $U\ VM$, the times of the virtual memory based implementation with checking done only by GDB; $U\ B$, times of the HP PA B-bit based implementation with checking done only by GDB; $K\ VM$, the times for the virtual memory based implementation that uses kernel checking; $K\ B$, the times for the HP PA B-bit based implementation that uses kernel checking.

4.3 Global Variables

Figures 4.1, 4.2, 4.3, and 4.4 indicate the median, maximum and minimum times in seconds it took for each of the global tests to run using the different implementations. The data used to create all of the graphs in this thesis are found in tables in appendices B through E. Also note that the graphs use a log scale.

The times for running the programs without any watchpoints give an indication of how much overhead is incurred when watchpoints are set. The times for single stepping show that this implementation is so slow that to be useful, the user must

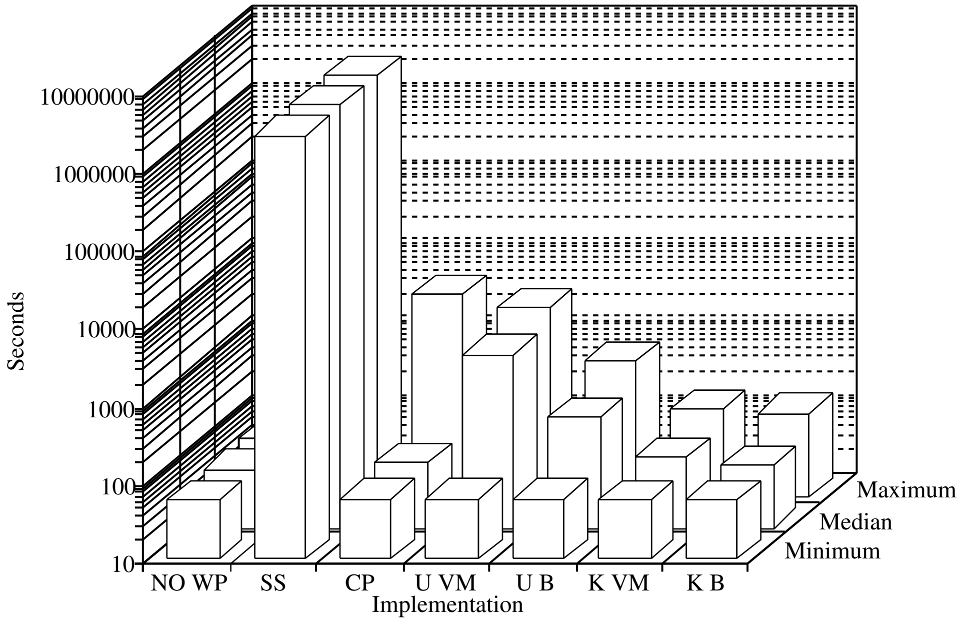

Figure 4.1. Eqntott global test results

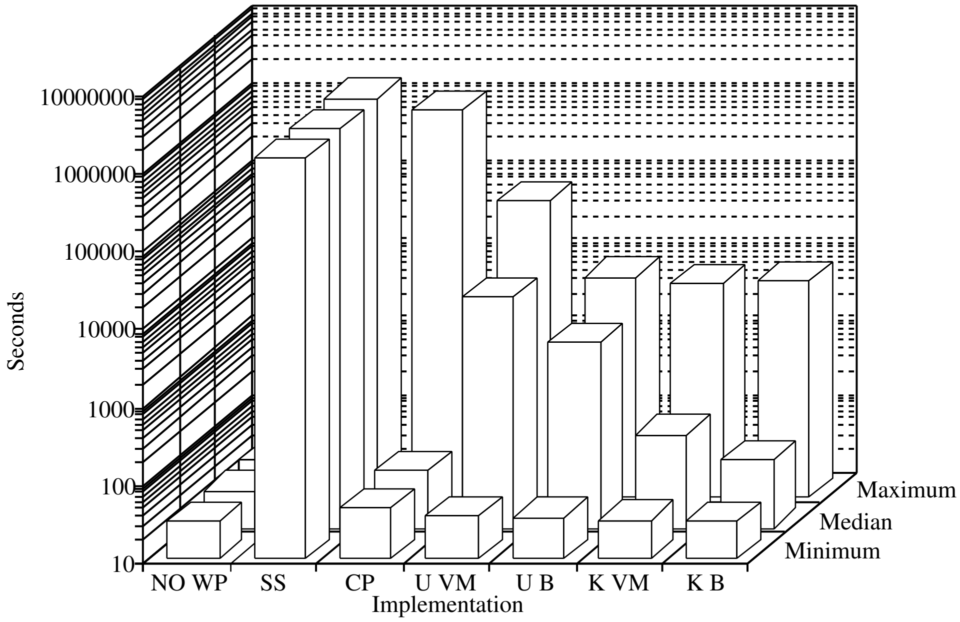

Figure 4.2. Espresso global test results

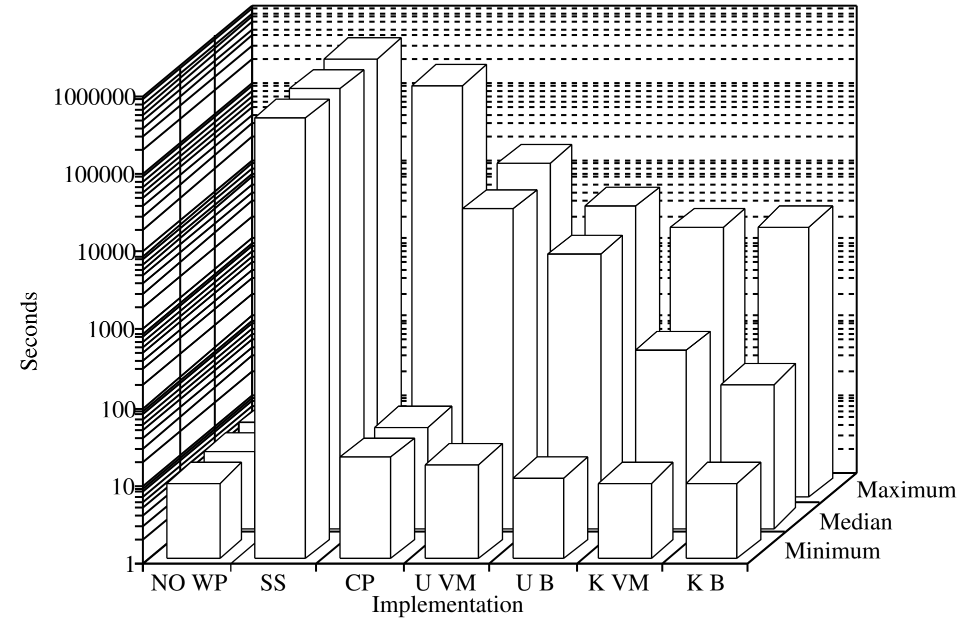

Figure 4.3. Xlisp global test results

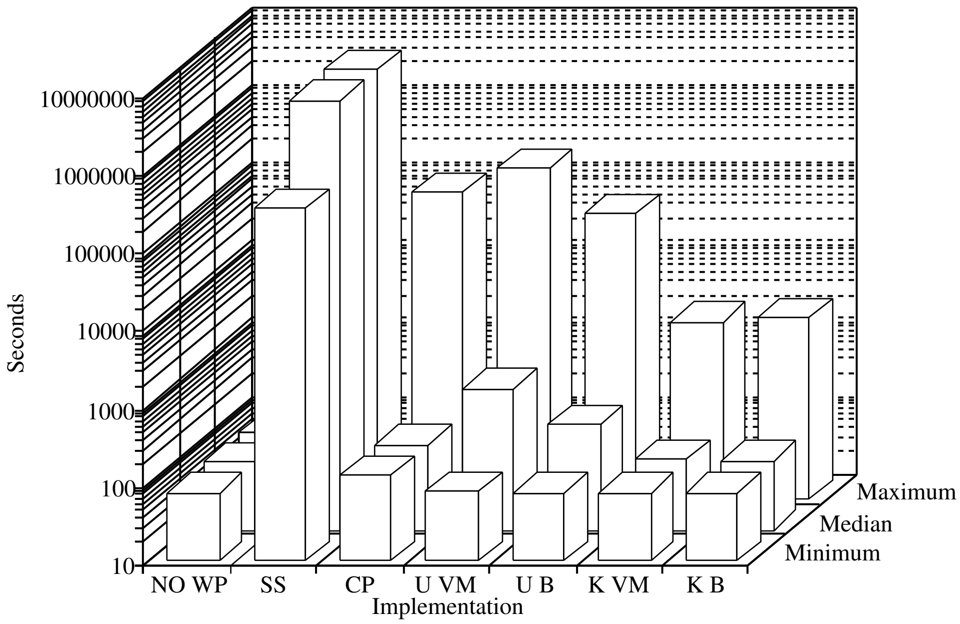

Figure 4.4. Gcc global test results

limit how they are used. A program that normally takes a minute to run takes a over month to run while single stepping.

The code patching implementation performs well most of the time. Its median times are among the best of all of the solutions. The very slow times presented in the maximum times table occur when watching variables that are written thousands and even millions of times. These particular times are slower than any tests for any implementation except single stepping. These tests are slow because the implementation of inserting and removing watchpoints is very slow in the code patching implementation. Since every time the program stops the watchpoints are removed, and every time the program starts they are inserted, the high overhead of these operations adds up quickly when there are many stops. Nevertheless, in an interactive environment, this overhead is spread over many stops and is not too noticeable to the user. If, however, the environment is noninteractive or the watchpoint is conditional, then it becomes imperative to reduce the overhead of inserting and removing watchpoints. It is interesting to note that the code patching implementation has the worst minimum times. This is due to the slowdown caused by executing the additional code in the debuggee.

The implementation using virtual memory without kernel checking performs considerably worse than the code patching. Furthermore, it is not just a few tests that are slow, it is most of them, although a few do perform quite well. This poor performance is especially true for the xlisp and espresso programs. With these two programs, most of the global variables reside on one page and there are a few variables on that page that are written to over a million times. With eqntott, again most of the variables reside on one page, but that page is not written as frequently, but a few variables still account for most of the writes. Gcc, on the other hand, has many more variables that are spread out over several pages, and some of the pages are rarely written whereas others are written extensively. As a result, many of the gcc tests performed quite well while others performed poorly.

The performance of the HP PA B-bit based implementation is three to four times faster than the virtual memory based implementation. As expected, the improved performance is uniform—each test shows comparable improvement. Nevertheless the performance is considerably worse than the code patching based implementation.

Kernel checking improves the performance of the virtual memory based implementation by a factor of 50 to 70, while improving the performance of the HP PA B-bit based implementation by a factor of 40 to 50. The overhead of the virtual memory based implementation with kernel checking is approximately three times greater than the overhead of the HP PA B-bit based implementation with kernel checking, the latter being about par with the code patching based implementation.

The results indicate that code patching and both versions using kernel checking perform extremely well. It is difficult to say definitively whether code patching or kernel checking performs better because some of the tests perform better with code patching while others perform better with kernel checking. However, based upon how the different implementations perform on each test, it can be concluded that code patching performs better on small programs whose data resides only on a few pages, especially if those pages are written often; whereas the kernel checking based implementations perform better on large programs with the data spread out over several pages, especially if the memory that is being watched resides on a page that is not written to millions of times.

Nevertheless, it is apparent that the virtual memory and B-bit based implementations without kernel checking perform worse than the code patching and kernel checking based implementations yet considerably better than the single stepping based method GDB originally used. The questions now become what is the true measure of their performance and how acceptable is it. The nonkernel checking methods perform far better on the gcc and eqntott tests perform much better than on the espresso and xlisp tests. There may be many large programs like gcc that do not have the concentration of data that hinder the performance of these implementations and perform better than the small write intensive programs. Also, small programs that are not write intensive like the benchmark programs may perform well with these implementations. Still, the worst case behavior exists and may be common in real programs.

4.3.1 Work-arounds

At first glance the performance of the virtual memory and B-bit based implementations might seem disappointing. However, much of the poor performance is due to a few variables that are written many times. The writes to these variables ruin the performance of watching all other variables on the same page. Most of the offending variables are found in loops that execute over a million times. By simply stopping the program before the loop, disabling the watchpoint, continuing until the end of the loop, checking by hand if the value of the expression has changed and then re-enabling the watchpoint before continuing, almost all of the overhead is eliminated. This is analogous to a user controlled checkpointing solution to the problem. Of course this is not the perfect solution, as it requires more effort on the part of the user and in some cases may not be good enough to find the problem. Nevertheless, though annoying, many users will be willing to do this to improve performance, as evidenced by the current state of watchpoints—to derive any use from single stepping based watchpoint implementations, the user is forced to use this type of trick much more extensively.

Unfortunately, not all of the offending variables are in large loops. It is possible to isolate many of the remaining problem variables by altering the executable in such a way that they are put on a page of their own. This may be asking too much of the average programmer to do in debugging his program, but still may be a valid work-around for someone who understands what is happening.

To determine the amount of overhead of the VM based schemes that can be avoided, I evaluated versions of the three SPEC programs that normally have poor performance using the two work-arounds described above. Figures 4.5, 4.6, and 4.7 indicate the median times in seconds it takes for the global tests to run with and without the work-arounds.

The results indicate that the overhead of the median case is drastically reduced with the work-arounds. There is little improvement in the maximum and minimum times. The maximum times represent tests in which the watched expression changes millions of times, so the work-arounds are not applicable to them, while

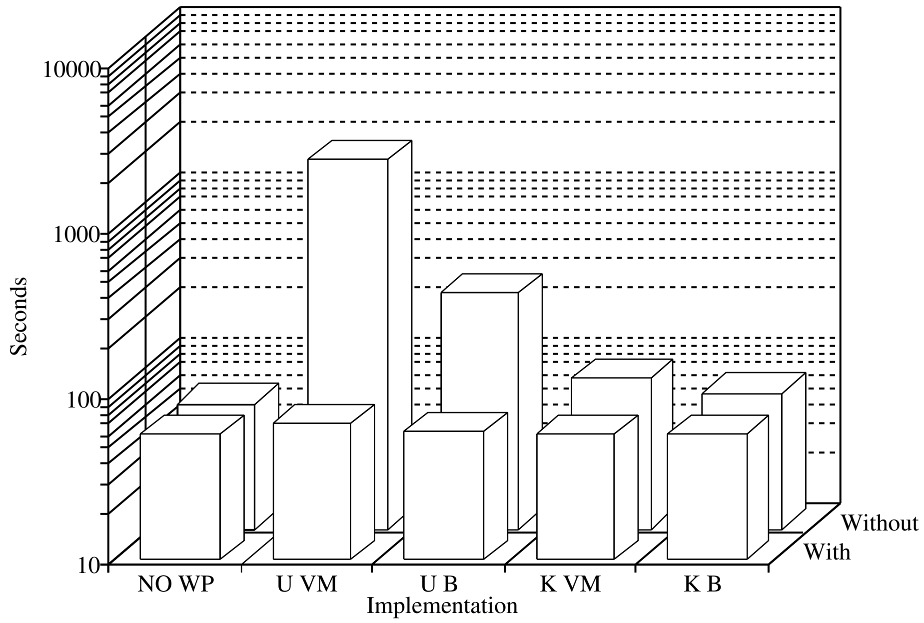

Figure 4.5. Eqntott global test results with work-arounds

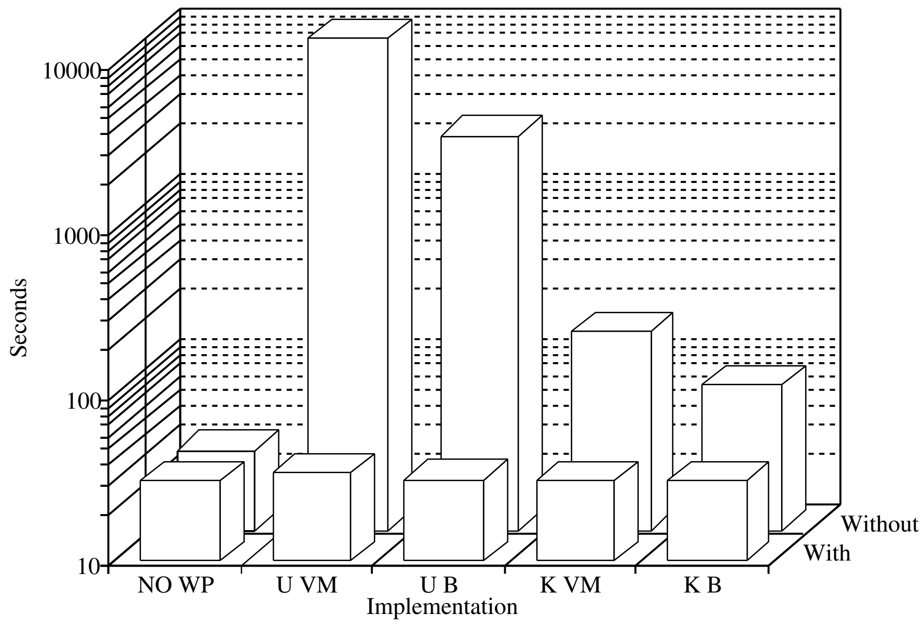

Figure 4.6. Espresso global test results with work-arounds

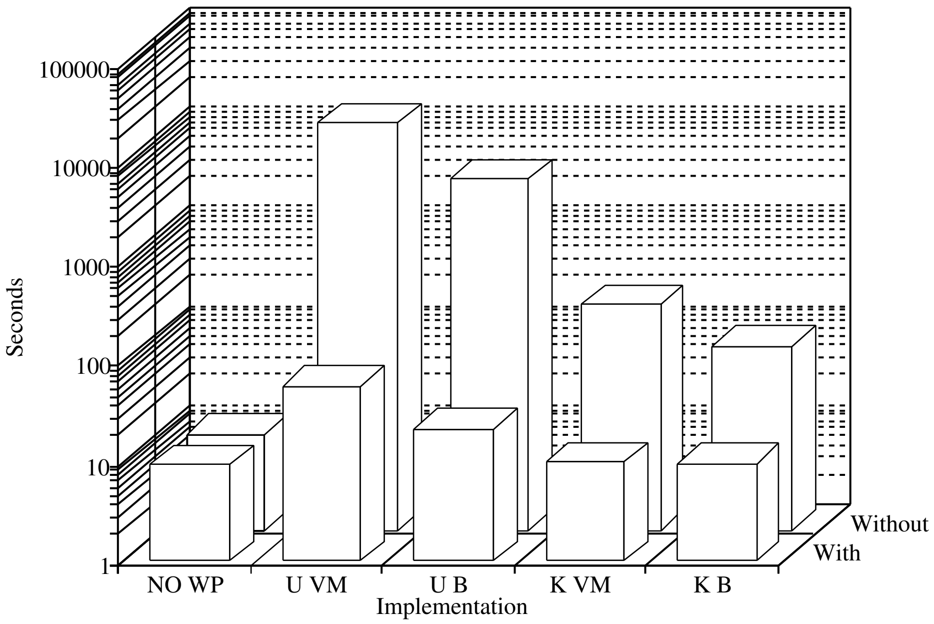

Figure 4.7. Xlisp global test results with work-arounds

the minimum times have little room for improvement. With the work-arounds the virtual memory and B-bit based implementations perform as well as or better than any implementation studied. Though these implementations will at times perform poorly, with a little effort on the part of the user these implementations can be made useful. Furthermore, they are much better than the single stepping method, so if other implementations are not feasible, these implementations are a viable alternative.

4.3.2 Histograms

Figures 4.8 through 4.11 present histograms of the global test results, which give a more detailed picture of how each implementation performs. The y-axis represents the number of tests that finished in the indicated range of time.

The histogram of the running times for the eqntott global tests shows that only the virtual memory and B-bit based implementations without the kernel checking have a significant number of tests that perform poorly. Nevertheless, these implementations perform quite well on almost half of the tests, and with the work-arounds, they almost always perform well.

The histogram of the running times for the espresso global tests indicates that the code patching based implementation performs a little better than the B-bit version with kernel checking, which performs better than the virtual memory based implementation with kernel checking. All of these options perform considerably better than the virtual memory and B-bit implementations without kernel checking. However with the work-arounds, all of the virtual memory and B-bit based implementations perform better than code patching. The histogram also reveals that there are just two tests that perform poorly with the work-around, these tests correspond to watching variables that are written many times, one of which is written millions of times.

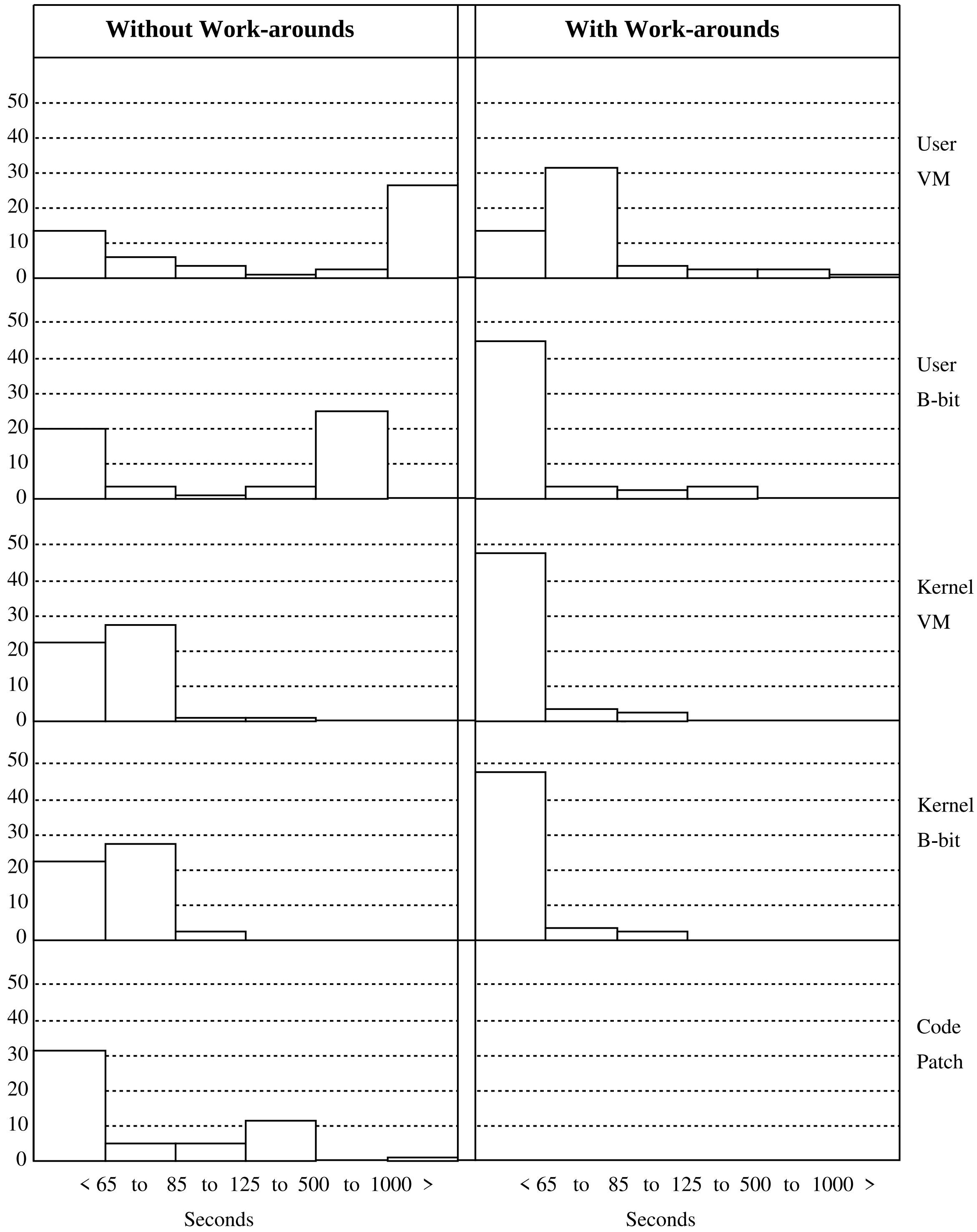

Figure 4.8. Histograms of eqntott global test results

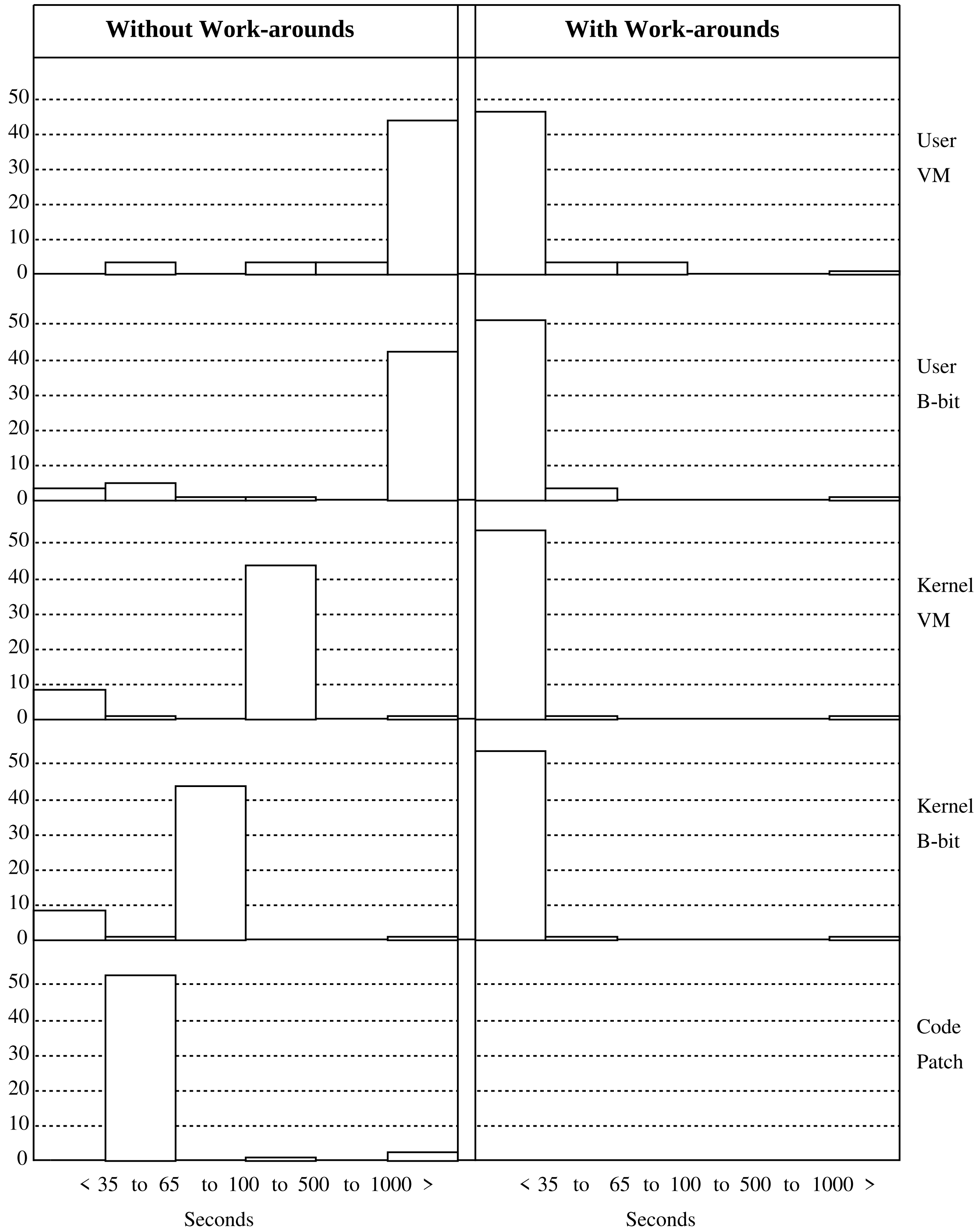

Figure 4.9. Histograms of espresso global test results

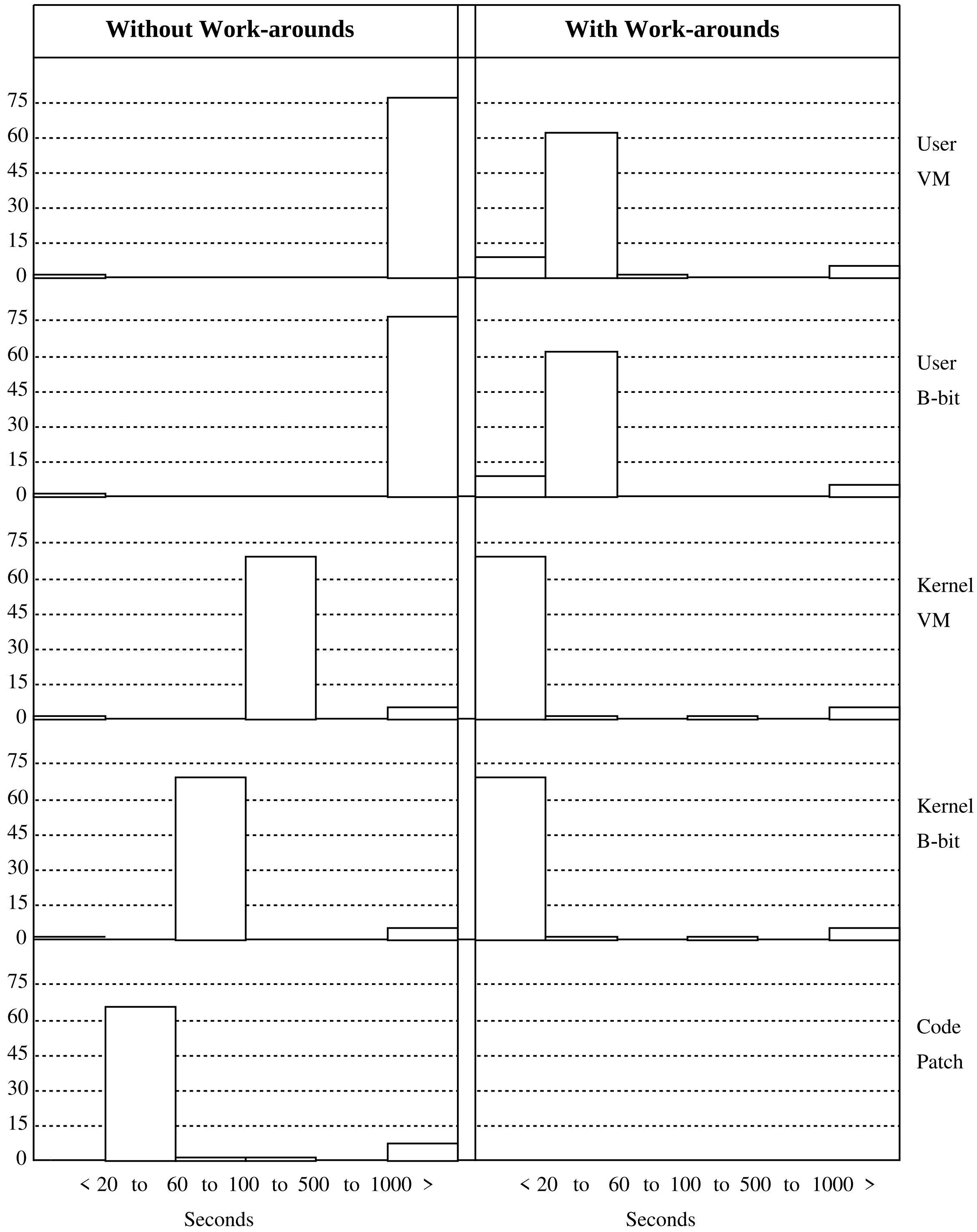

Figure 4.10. Histograms of xlisp global test results

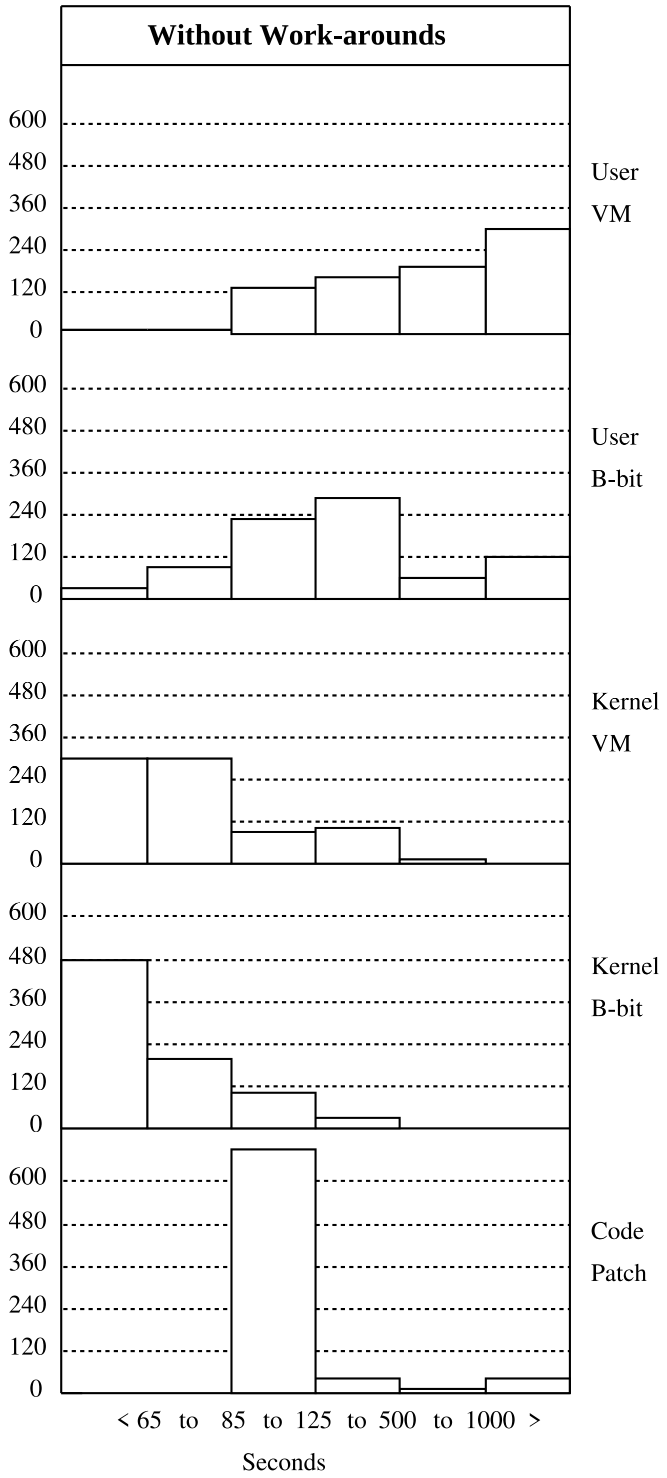

Figure 4.11. Histograms of gcc global test results

The histogram of the running times for the xlisp global tests is almost identical to the histogram of the running times for the espresso program, except that there are more variables that are written extensively.

The histogram of the running times for the gcc global tests indicates that the virtual memory and B-bit based implementations with kernel checking perform better than the code patching based implementation. The virtual memory and B-bit based implementations without kernel checking again perform worse than the others, however they do not perform as poorly as they did with the other programs.

The histograms are consistent with the data and conclusions previously given in section 4.1, so the conclusions will not be repeated here. The main purpose of including the histograms is to expose the variance across test runs, which is marked by the median values plotted in Figures 4.1 through 4.7.

4.4 Heap Variables

Figures 4.12 through 4.15 show the median, maximum and minimum runtimes in seconds of the tests involving the heap variables.

The results of the heap variable tests are very similar to the results of the global variable tests except that the heap tests are generally slower, because the general expression used to watch a heap variable usually requires more than one location to be watched. For example, if GDB were to watch the expression $*p$, changing the value of p as well as changing the value of what p points to changes the value of the expression, so both locations must be watched. If one is truly interested in watching the memory in the heap, it is possible to construct an expression that watches that memory explicitly. However, my tests all use the use the general symbolic expression.

There are a few results that appear to be better than those from the global tests, however this is because the pointers to the heap locations happen to reside

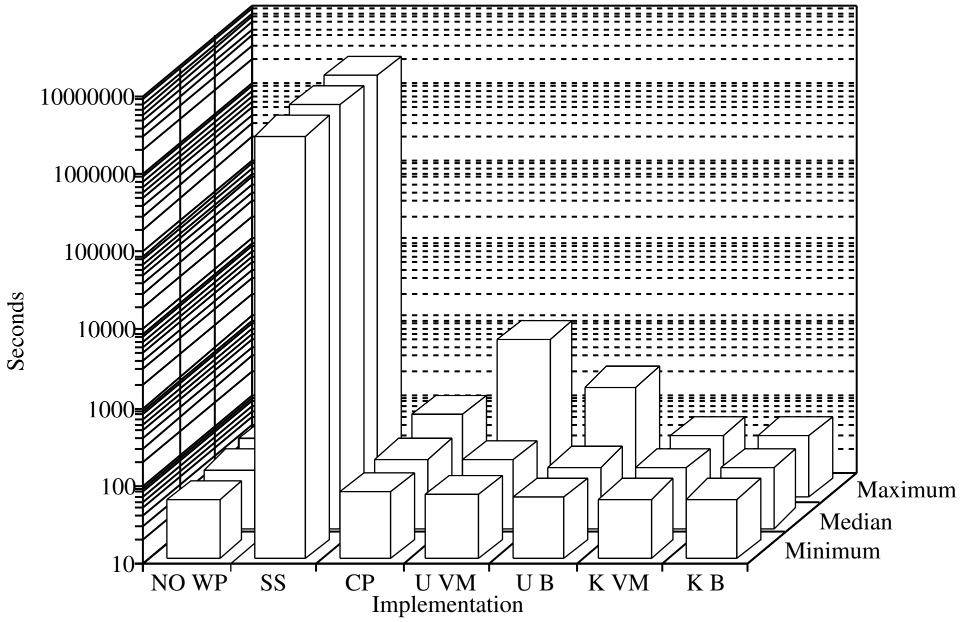

Figure 4.12. Eqntott heap test results

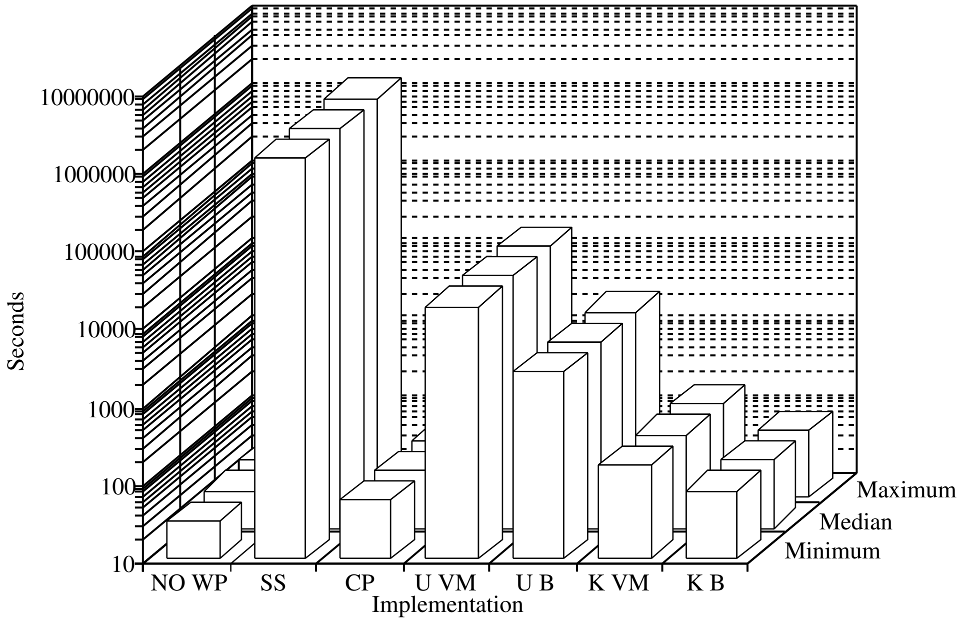

Figure 4.13. Espresso heap test results

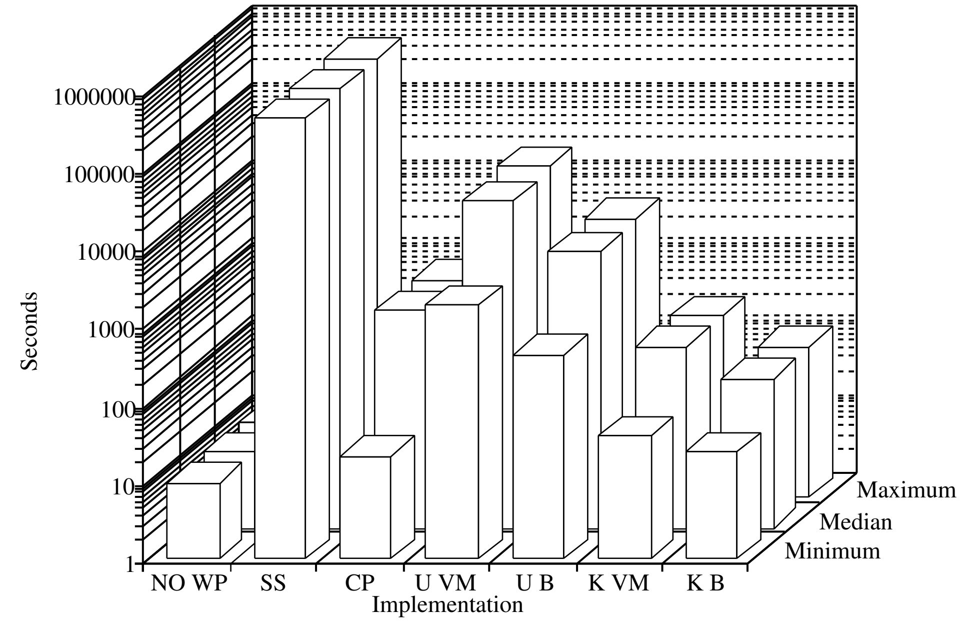

Figure 4.14. Xlisp heap test results

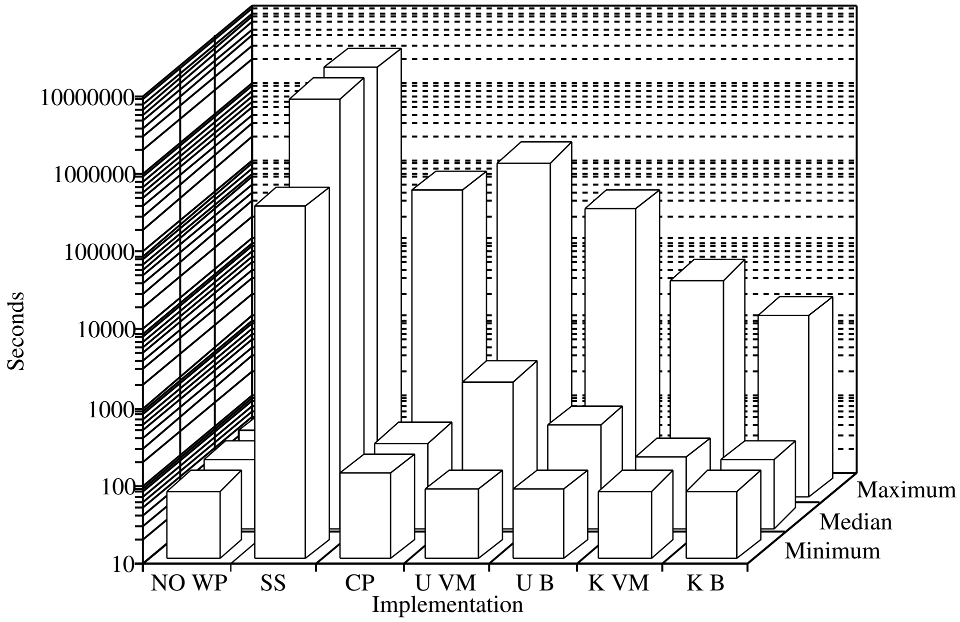

Figure 4.15. Gcc heap test results

on pages that are not written often. Nevertheless, the results from the heap tests mirror the results from the global tests quite closely. The only major deviations from the results of the global tests are that the results of the code patching heap tests of the xlisp program are worse than the corresponding results of the global tests. The deviation arises because most of the heap items in this program are written to thousands of times, and therefore the overhead due to inserting and removing watchpoints becomes a factor.

Like the results of the global tests, the results of the heap tests show that the code patching and kernel checking implementations perform the best, while the virtual memory based implementations without kernel checking perform worse. These implementations perform particularly poorly on the xlisp and espresso programs, but not so poorly on the gcc and eqntott programs. Again, the performance is poor when watching an expression that causes a page that is written extensively to be protected and is much better when the protected pages are not written much.

4.4.1 Work-arounds

As with the global test, the performance of the heap tests can be improved with work-arounds. Specifically, almost all of the overhead in the heap tests is due to protecting the page on which the pointer resides. Therefore, improvements in performance can be achieved not only by the work-arounds used to improve performance of the global tests but also by watching the memory of the object in the heap explicitly. Figures 4.16 and 4.17 shows the median times of the heap tests of xlisp and espresso programs with and without the work-arounds.

In the case of the espresso program most of the overhead is eliminated because almost all of the overhead is due to protecting the page on which the pointer resides. With the xlisp program 90% of the overhead is eliminated, but there is still a large amount of overhead due to protecting the page in the heap and the frequent stops

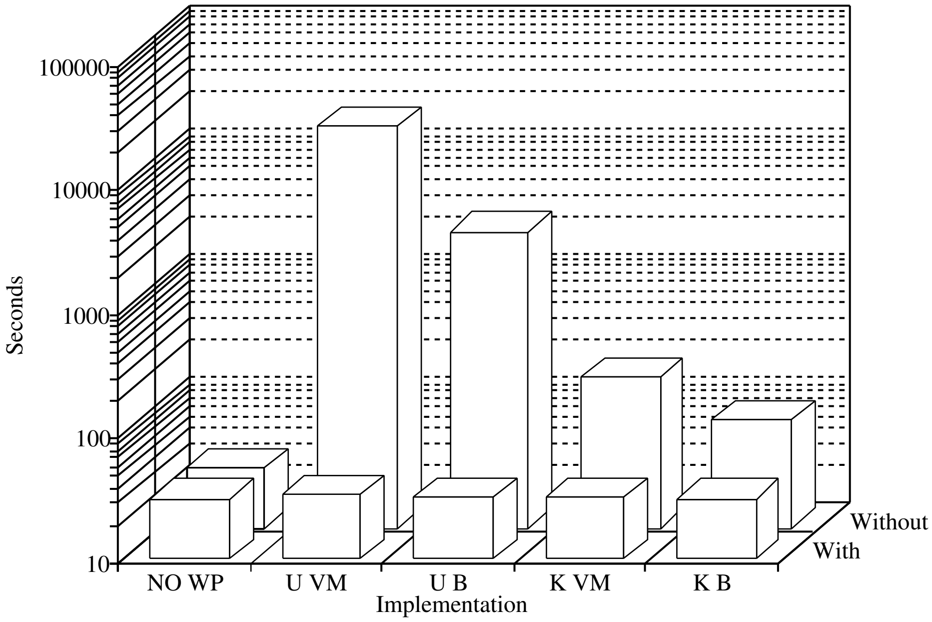

Figure 4.16. Espresso heap test results with work-arounds

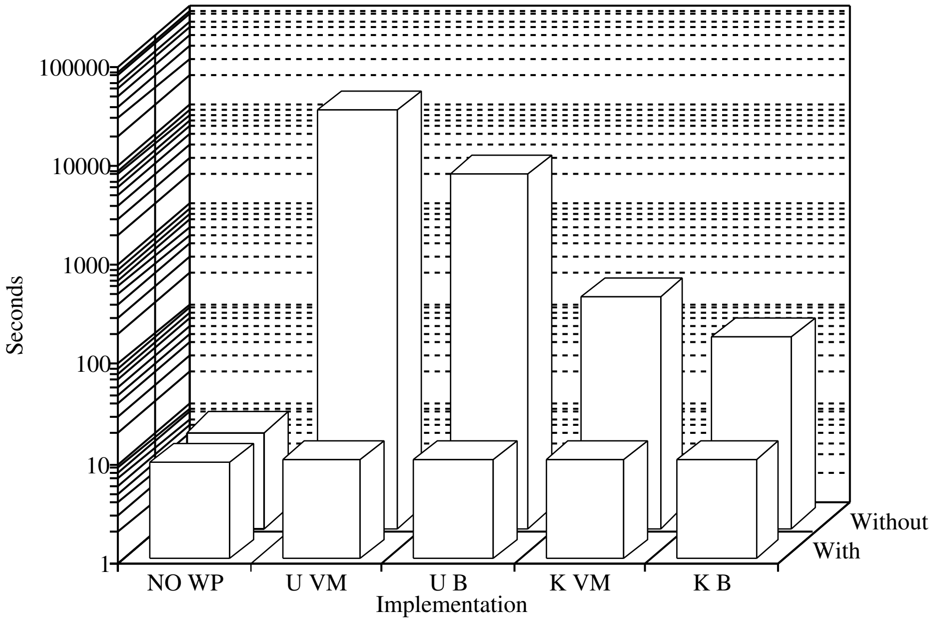

Figure 4.17. Xlisp heap test results with work-arounds

due to the value of the expression changing.

4.4.2 Histograms

Figures 4.18 and 4.19 present histograms of the heap test results from the xlisp and gcc programs, which give a more detailed picture of how these implementation perform.

Histograms of the running times for the eqntott and espresso heap tests are not presented because because there are only three and one data points respectively, thus no additional information could be given than was given with the maximum, median and minumum results.

However, as a reminder, these are the results of the eqntott heap tests: the code patching based implementation performed acceptably on all three tests; the virtual memory and B-bit based implementations without kernel checking performed very well on two of the tests and poorly on one of the tests; the implementations with kernel checking performed very well on all three of the tests. These results are indicative of the fact that none of the heap items are written extensively; however one heap item does reside on a page that is written extensively.

Likewise, the results of the espresso heap test are that the code patching based implementation performed well on the test whereas virtual memory and B-bit based implementations performed very poorly without kernel checking and poorly with kernel checking. However, with the work-arounds they all performed very well. These results can be attributed to the fact that the pointer to the heap item resides on a page that is written extensively, but neither the heap item nor the page it resides on are written very much.

The histograms of the running times for the xlisp heap tests indicate that the code patching based implementation performs poorly most of the time; the virtual memory and B-bit based implementations without kernel checking perform very

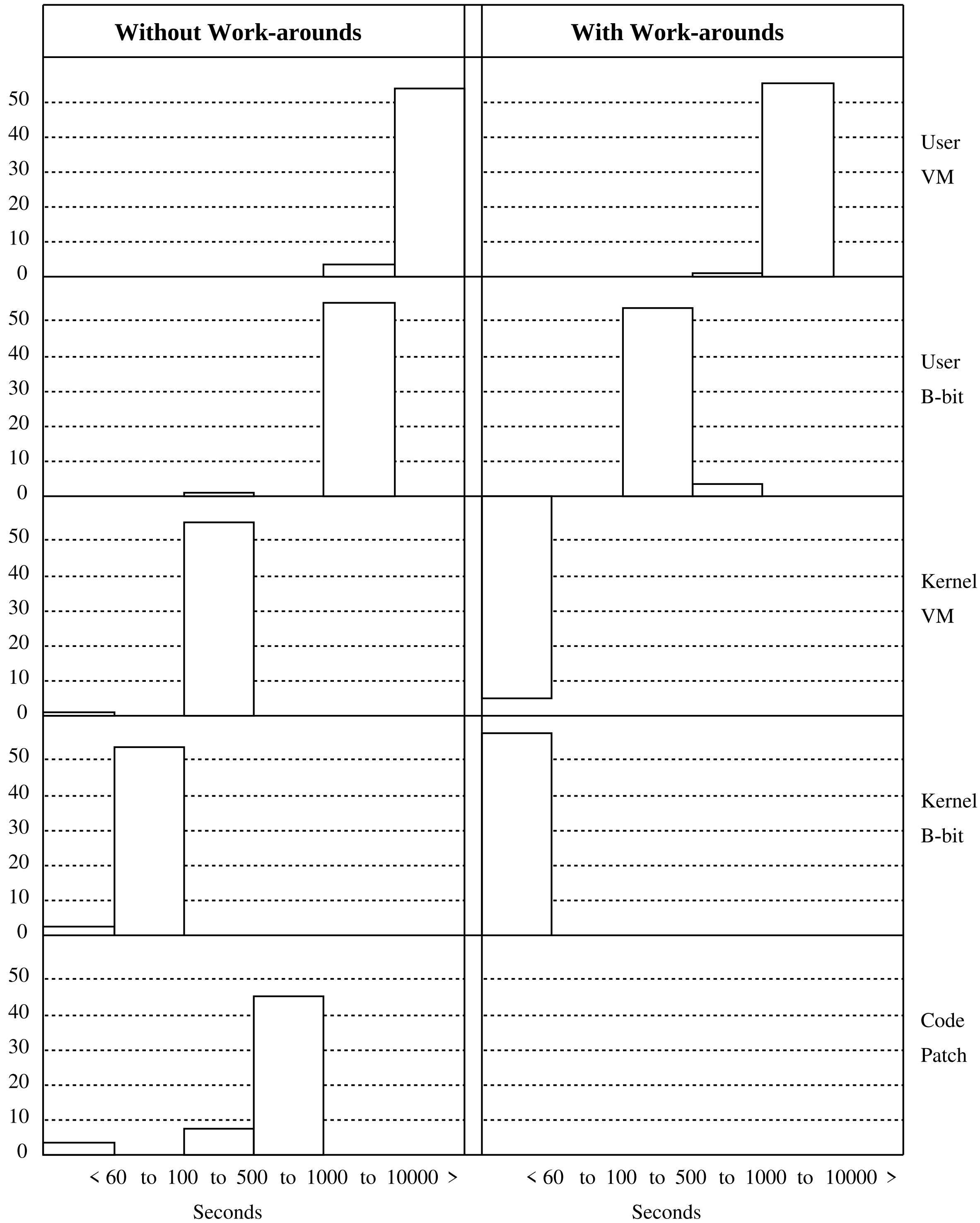

Figure 4.18. Histograms of xlisp heap test results

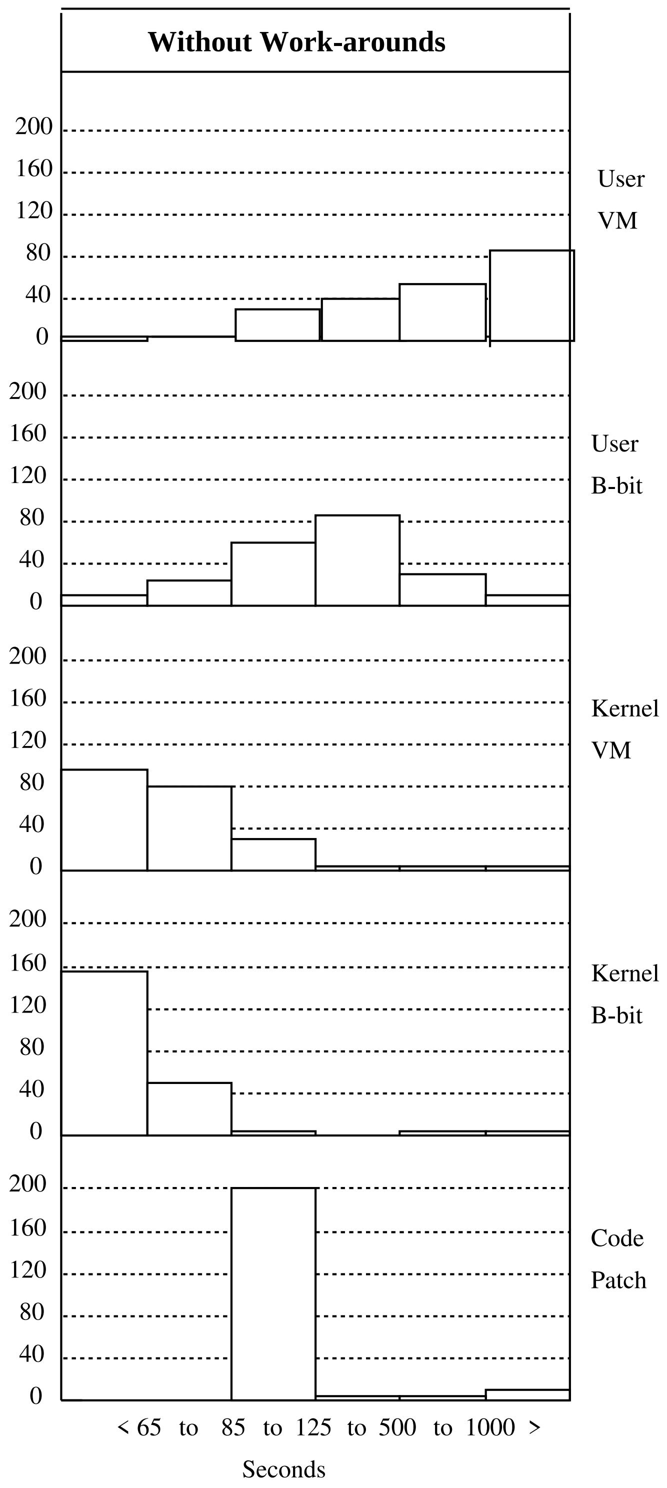

Figure 4.19. Histograms of gcc heap test results

poorly; the implementations with kernel checking perform well. The work-arounds reduce the overhead by an order of magnitude. These results can be attributed to the fact that not only are the heap items written extensively, but also the pages on which the heap items and their pointers reside. The kernel checking based implementations perform best on these particular tests even though there are many writes to the protected pages because the code patching based implementation incurs the heavy overhead of inserting and removing the watchpoints every time the value of the expression changes.

The running times for the gcc heap tests have greater variance than the running times for the other programs tested. This is especially true for the virtual memory and B-bit based implementations, which is indicative of the fact that gcc uses more pages for its data than the other programs and some of the pages are written more than others. Nevertheless, most of the tests of the virtual memory and B-bit based implementations with kernel checking perform very well and those without kernel checking perform adequately. The code patching based implementation generally performs worse than the kernel checking based implementations and better than the implementations without kernel checking.

The results from the heap test confirm the previous conclusions that the code patching based implementation performs well except when the value of the expression changes often, in which case it performs very poorly. The virtual memory and B-bit based implementations with kernel checking perform very well when the protected pages are not written extensively and still perform well when they are. However the virtual memory and B-bit based implementations without kernel checking only perform well when the protected pages are not written extensively and perform poorly when they are.

4.5 Local (Stack) Variables

Figure 4.20 shows the results of the stack test, which watches a memory location on the stack for the duration of the program. In this figure, the implementations on the x-axis are ordered from worst to best performance in order to make visible all of the bars in the graph. As one can see, the code patching implementation performs well on the stack test for all programs. The virtual memory based implementations without kernel checking perform very poorly on the eqntott and espresso tests and poorly on the xlisp and gcc tests. The versions with kernel checking perform well on the xlisp and gcc tests but perform poorly on the eqntott and espresso tests. The poorer performance of the virtual memory and B-bit based implementations can be attributed to fact that the stack is generally written more often than other data. Again, the B-bit implementations perform better than the simple virtual memory implementation as is expected.

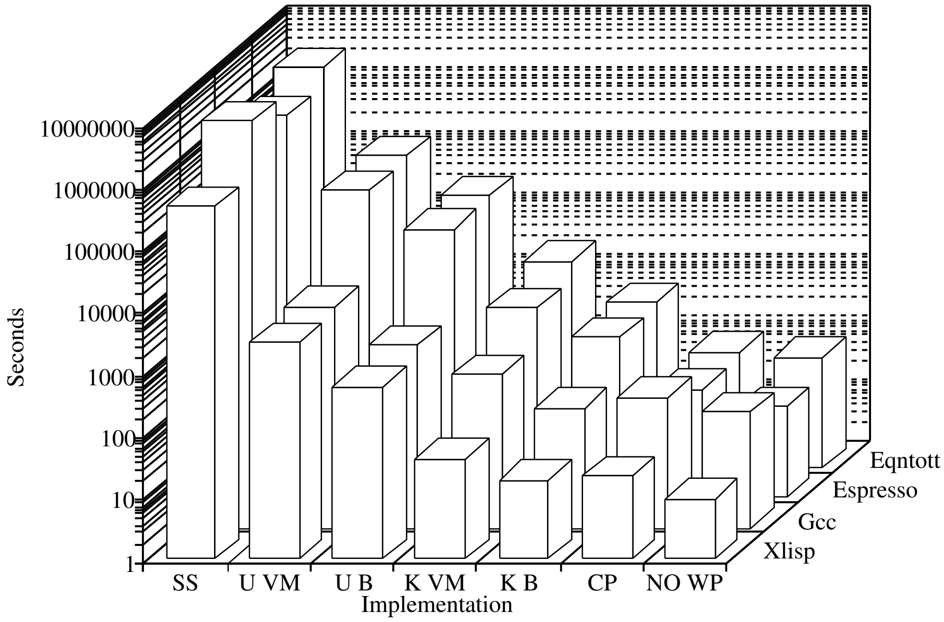

Figure 4.20. Results of the stack test

4.6 Analysis

In this section a number of specific cases are analyzed in detail to determine how they perform with the different implementations. I have chosen a representative group of tests that show the strengths and weaknesses of the implementations.

From the eqntott program, I examine the performance of the different implementations when watching the global variables *noutputs* and *yyps* (see Table 4.1). The variable *noutputs* is written 29 times but is on a page that is written over 700,000 times. The variable *yyps* is written 372 times but the page is written less than 10,000 times. The program normally takes 57 seconds to run.

The eqntott program takes 58 seconds to run after it has been patched with no variables being watched, which means that in the case of the eqntott program, the overhead for patching the code is small, less than 2%. In the code patching implementation, each stop caused by a write to the variable takes approximately one fifth of a second. Because the variable *yyps* is written more than the variable *noutputs*, the test that watches *yyps* takes longer than the test that watches *noutputs*.

In the virtual memory implementation the relative performance of the two tests is reversed from the code patching implementation. This is because the performance of the tests is dependent upon the number of writes to the page and not just the number of writes to the variable. The performance characteristics of the HP PA break bit implementation are similar to the virtual memory implementation, as

Table 4.1. Results of example eqntott tests in seconds.

	code patch	User VM	User B-bit	Krn VM	Krn B-bit
noutputs	64	1710	519	81	66
yyps	133	61	59	58	58

expected, with only one third of the overhead. As always, kernel checking greatly reduces the overhead.

From the xlisp program, the global variable *xlstack* and the heap variable $*true$ are monitored (see Table 4.2). The variable *xlstack* is written 1,469,060 times and resides on a page written over 5,000,000 times. The variable *true* resides on the same page as *xlstack* and points to memory on a page that is written about 500,000 times. The memory associated with $*true$ is written 2424 times. The xlisp program normally takes 9 seconds to run.

The poor performance of the code patching implementation on these tests is due to the number of times that the variables are written. Performance could be improved with a more efficient mechanism of inserting and deleting the watchpoints as described earlier.

Again the performance of the virtual memory implementation is dependent mostly upon the number of times the pages of the watched memory is written, although the variable *xlstack* is written so many times that it too has some effect on the performance. Watching the memory associated with $*true$ directly is more than 10 times faster than watching the expression $*true$ because the page on which the variable *true* resides is written so often.

The performance of the HP PA break bit implementation is four times better than the virtual memory implementation when watching the expression $*true$ and two times better when watching the expression *xlstack*. The performance for

Table 4.2. Results of example xlisp tests in seconds.

	code patch	User VM	User B-bit	Krn VM	Krn B-bit
xlstack	206000	13800	5430	2930	3020
$*true$	642	16300	3680	211	83
mem of $*true$	642	1440	334	33	21

xlstack does not improve as much because of the dependence upon the number of times which it is written. Again, directly watching the memory of $*true$ is more than 10 times faster than watching the expression.

As always, kernel checking greatly improves performance. This time, the performance of the *xlstack* test is almost entirely dependent upon the number of writes to the variable and not to the page as the kernel checking considerably reduces the overhead of checking the false hits.

In gcc the global variable *flag_cse_skip_blocks* is monitored (see Table 4.3). The expression *flag_cse_skip_blocks* changes one time, although it is written many more times, as it is repeatedly cleared to zero. The page on which it resides is written about 150,000 times. The program normally takes 74 seconds to run.

This test is interesting because it contradicts the assumption that control only returns to the debugger when the value of the expression has changed. This is not always the case as this test illustrates by the poor performance of the code patching implementation. There are also other cases where control is returned to the debugger without the expression changing. This is most evident when the simple mechanism GDB uses to determine which addresses to watch includes addresses that do not need to be watched. This occurs when watching elements of an array or fields of a structure, in these cases the whole array or the whole structure are actually watched.

The code patching based implementation and the kernel checking based implementations could be changed so that they check for the value of the watched

Table 4.3. Results of example gcc tests in seconds.

	code patch	User VM	User B-bit	Krn VM	Krn B-bit
flag_cse_skip_blocks	1130	3780	975	123	91

memory changing instead of just written which would improve the performance of this last test; however the additional overhead on every check would increase, so the general performance would worsen.

The six tests analyzed in this section illustrate why each of the implementations perform as they do. Code patching, as it is implemented, is dependant upon the number of times the program stops. The virtual memory and B-bit based implementations are dependant upon the number of times to which the protected page is written. Also shown in the analysis are some cases of unnecessary overhead caused by stopping when the expression has not changed and protecting pages in which the user may not be interested.

4.7 Comparison

Most of the watchpoint research and development being done lately use the results from a simulation study done at the University of California, Berkeley[24] as their basis. Table 4.4 is a comparison between the timings of the basic watchpoint operations as measured in the implementations discussed in this paper and the corresponding numbers put forth in the simulation study.

The simulation study claimed that inserting and removing watchpoints in a code patching implementation could be performed much faster than the GDB implementation shows. There are two reasons for the difference. First, the sim-

Table 4.4. Comparison between simulation study and real implementation in microseconds.

	Sim CP	Sim VM	code patch	User VM	User B-bit	Krn VM	Krn B-bit
insert	22	401+	70000	540	540	540	540
remove	22	401+	70000	540	540	540	540
check(miss)	2.75	564	2	2400	660	34	13
check(hit)	2.75	564	140000	3500	2400	2400	2400

ulation study completely ignored the communication cost between the debugger and the debuggee which most certainly will be the biggest factor. Second, the GDB implementation is grossly inefficient. Still, if the current *ptrace* is used as the method of communication, 500 microseconds is the best that could be achieved. The numbers for the virtual memory based implementations are close to what the simulation study expected, but for the wrong reasons. The simulation study attributed most of the overhead of inserting and removing watchpoints to protecting and unprotecting the page; whereas in actuality, almost all of the overhead comes from *ptrace*. Also, the table shows the cost for a virtual memory implementation in the simulation study to be 401+. This is because the simulation study assumes the data structure that keeps track of the watchpoints is in the debuggee and incurs additional cost to protect it. However, it makes little sense to put the data structure in the debuggee in a virtual memory implementation so this cost is not included.

In comparing the simulation study to the real implementation, the numbers for checking writes that miss, that is that write to memory that is not being watched, are similar in the code patching case. In the virtual memory case the simulation study underestimated the cost. However, the cost becomes comparable with the use of the HP PA B-bit, and becomes much faster when the operating system does the checking. Again, the major differences in the cost of this operation are due to communication or context switching. The code patching has no additional overhead due to communication or context switching. In the HP PA B-bit implementation with kernel checking, control is given to the kernel to check the address of the write and then returned to the debuggee. In the virtual memory implementation with kernel checking there is an additional overhead to unprotect the page, emulate the instruction, and reprotect the page. In the cases without kernel checking, control is returned to the debugger. In the HP PA B-bit based implementation it only takes one *ptrace* call to continue while the virtual memory based implementation requires four: one to unprotect the page, one to single step, one to reprotect the page and one to continue. Clearly *ptrace* is the bottleneck to the implementations without kernel checking. Significant improvements could be made with more a efficient mechanism of communication. Perhaps *procfs* will perform better than *ptrace*. Other operating systems like Mach could possibly perform much better: the debugger could use the task port of the debuggee to protect and unprotect pages without a system call.

The simulation study also underestimated the cost of checking writes that change the expression. Again, they completely ignore the cost of returning to the debugger and also the cost of removing and reinserting the watchpoints, which, as been discussed, is very expensive.

As an exercise, it would be interesting to compare how well the code patching implementation would do if inserting and removing watchpoints were as efficient as the virtual memory based implementations. If that were the case, the only significant difference between the performance of the implementations would be the time to check writes that miss the watched memory. Remembering that the code patching implementation checks all of the writes and the virtual memory based implementations check only the writes to the protected page, then it can be concluded that the virtual memory based implementation without kernel checking will only perform better than code patching when the writes to the protected page account for less than .08% of the total writes. Likewise the HP PA B-bit based implementation without kernel checking will perform better if less than .3% of the writes are checked; and with kernel checking the virtual memory and HP PA B-bit based implementations will perform better when less than 6% and 15% of the writes are checked, respectively. In other words, if one is to assume uniform distribution of the writes amongst the pages and only one page is protected, then the program would need to use 1200 pages for the virtual memory based implementation without kernel checking to perform better than the code patching based implementation; while it would need to use 330 pages for the HP PA B-bit based implementation without kernel checking to perform better. With kernel checking, the program would need to use only 17 and 7 pages for the virtual memory and HP PA B-bit implementations to perform better, respectively.

This exercise has made clearer what has already been revealed in this thesis, that is that code patching and the virtual memory based implementations with kernel checking typically perform about the same, while the virtual memory based implementations without kernel checking typically perform considerably worse. Furthermore it has quantified how expensive communication between debugger and debuggee is, and in comparison how inexpensive communication between program and operating system is when that program faults and the operating system is able to handle the fault and return control to the program.

CHAPTER 5

CONCLUSIONS

Watchpoints are a special type of breakpoint in which users of a debugger can specify that the program stop when the value of a given expression changes. Fast watchpoints are a great benefit to many users of the debuggers that have them. Watchpoints are the ideal way of finding certain types of bugs; unfortunately, watchpoints as they are currently implemented in many debuggers are so slow that they are unusable in any practical sense. A number of techniques for improving watchpoint performance have been proposed. A few of the most promising schemes were implemented in GDB to evaluate them in terms of speed, portability, applicability, ease of implementation, and user friendliness. Table 5.1 assigns letter grades to the different aspects of each approach.

The code patching implementation performed very well on average so it receives an A. Its only performance problem is the overhead of inserting and deleting watchpoints, which for the most part is not a serious problem since these operations

Table 5.1. Grading the approaches.

	Fast	Portable	Applicable	Easy	Friendly
Code Patch	A	B	A	C-	C-
Virtual Memory	D	A	B-	A-	A
HP PA B-Bit	C-	A	D	A	A
Kernel VM	A-	A	C	A-	A
Kernel B-bit	A	A	D	A	A
VM w/ work-around	B	A	B-	A-	C-
B-Bit w/ work-around	B+	A	D	A	C-

are usually not the bottleneck. If it were to become a serious problem, it could probably be remedied either by improving the mechanism to call inferior functions or by having gdb manage the data structure in the debuggee directly. Code patching receives a B for portability because there remains architecture specific code to be written for the tool that will patch the code. The greatest strength of the code patching implementation is its universal applicability. It can be implemented without any dependence upon operating system functionality or hardware support. The greatest weakness of the code patching implementation is that it requires the user's code to be altered. It some situations it may be completely unacceptable to alter the program. Not only could altering the program introduce new bugs and make others disappear, but it also could be a serious inconvenience to the user. Furthermore, depending upon the tool used to patch the code, it may even be impossible to patch the code, for example the tool used in this study requires the source code to do the patching. In addition, code patching is more difficult to implement than the other solutions. Not only does it require modification or creation of an additional tool to do the code patching, but also inserting and deleting watchpoints is not as straightforward as it is with the other implementations.

The virtual memory based implementation of watchpoints does not perform as well as the code patching implementation on most of my tests. However, there could be many programs that it would perform well on. Moreover, with a little wisdom in how watchpoints are used, the poor performance can be easily avoided on programs on which it normally performs poorly. This fact is indicated by the improved grade in speed with the work-arounds. An A is not given when the work-around is used because the user still must spend time finding the appropriate work-around. Nevertheless, the virtual memory based implementation is a great improvement upon the original implementation of watchpoints. One problem with the virtual memory implementation is that it requires support from the operating system, thus limiting its applicability. There is no reason why an operating system that allows a program to protect its own pages should not allow a debugger to protect pages of an inferior process; however, this capability is not present in many operating systems. All operating system developers should allow debuggers support this capability in their operating systems, as it only will improve the debugging capabilities of their systems. Overall, the virtual memory implementation is cleaner and easier to implement than the code patching implementation and user friendly in that it requires nothing additional from the user, whereas with code patching the user must worry about patching the code.

The HP PA break bit improves upon the virtual memory implementation by avoiding much of the overhead associated with stopping and continuing the program when a fault due to watchpoints occurs. It improves the performance by a factor of three to four and the implementation is cleaner and simpler. Having the break bit in the virtual memory system improves the HP PA's ability to do watchpoints efficiently and other computer architecture developers would be wise to follow HP's lead.

Adding support to the operating system kernel to check for watchpoints greatly improves the performance of the virtual memory and break bit based implementations. With kernel checking there are many times when these implementations perform better than the code patching implementation, although they still perform worse at other times. From a strictly performance standpoint it is difficult to say which performs better. From the debuggers point of view, the implementations with kernel checking are the easiest and cleanest to implement. However, kernel checking obviously requires the most operating system support. I understand the concerns that operating system developers would have in adding data structures to support watchpoints into the kernel, but if they are truly committed to having good debugging support for their system, then they should consider adding such support.

The fact that kernel checking improves the performance so greatly indicates how expensive it is in general for debuggers to communicate with the debuggee. The communication mechanism, *ptrace*, is very slow and any application that uses it extensively will have its performance hindered. Any improvements to the performance of *ptrace* or the communication mechanism will improve the performance of debuggers in general and especially those features, like watchpoints, that rely on this communication extensively. If *ptrace* is eternally doomed to poor performance, it may be wise to investigate other communication means. Some modern operating systems have better interprocess communication, such as allowing the processes to share pieces of memory. The use of alternative communication mechanisms may alleviate the *ptrace* bottleneck and greatly improve the performance of the virtual memory implementations without the need for additional kernel support.

This thesis provides several contributions over previous work to the study of watchpoints. Most of the previous and concurrent work in watchpoint development is based on the simulation study done at the University of California, Berkeley[24] which concluded that code patching was the best overall solution. Since then most work has concentrated on how to improve code patching techniques and how to implement these techniques into a debugger while mostly ignoring the virtual memory based implementation.

This thesis has concentrated mostly on improvements to the virtual memory based implementation and has shown that with these improvements, the virtual memory based implementation can perform just as well if not better than the code patching based implementation. There are three orthogonal aspects of the virtual memory based implementation that have been studied:

- additional hardware support for watchpoints via the HP PA break bit
- additional operating system support for watchpoints in the form of kernel checking
- distribution of data in debuggee via the work-arounds.

Furthermore, this thesis compares actual implementations in a real debugger, whereas the previous study compared *simulations* of implementations using an incomplete model. The result of this work has been incorporated into the most popular Unix debugger, GDB, and has led to a virtual memory based implementation on the Sparclite, monitor register based implementations on the Intel 386 and MIPS machines, and the break bit based implementation on the HP PA.

The implementation discussed herein could be improved by developing the following mechanisms: a more efficient mechanism for inserting and deleting watchpoints for the code patching implementation, a more selective mechanism for determining which memory locations to watch, and a mechanism to determine if a watchpoint needs to be removed and reinserted when the program stops and restarts.

In conclusion, the results indicate that those implementations that do much of the work outside the debugger and in the kernel or debuggee itself are fastest because they avoid much of the cost of context switches. Code patching appears to be most universally applicable and the VM based schemes are the easiest and least invasive to implement. All of the proposed solutions are much faster than the simplistic implementation that GDB and many other debuggers now use, and several are practical to implement and support.

APPENDIX A

CODE SAMPLES

A.1 A Simple C Program

```
struct int_point
{
  int r;
};

struct point_point
{
  struct int_point *q;
};

struct point_point *p;
struct point_point a[1000] = {0};
int i=1;
int result;


 main()
{
  int k;

  p = a;

  for (k = 0; k < 100; k++)
    {
      p->q = (struct int_point *) malloc(sizeof(struct int_point));
      i = k + i;
      p->q->r = i;
      p++;
    }
  result = a[99].q->r;
}
```

A.2 HP PA Assembly Language Translation

```
	.SPACE $PRIVATE$
	.SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=31
	.SUBSPA $BSS$,QUAD=1,ALIGN=8,ACCESS=31,ZERO,SORT=82
	.SPACE $TEXT$
	.SUBSPA $LIT$,QUAD=0,ALIGN=8,ACCESS=44
	.SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44,CODE_ONLY
	.IMPORT $global$,DATA
	.IMPORT $$dyncall,MILLICODE
; gcc_compiled.:
	.EXPORT a,DATA				; the variable a[]
	.SPACE $PRIVATE$
	.SUBSPA $DATA$

	.align 4
a
	.word 0
	.blockz 3996
	.EXPORT i,DATA				; the variable i
	.align 4
i
	.word 1
	.IMPORT __main,CODE
	.IMPORT malloc,CODE
	.SPACE $TEXT$
	.SUBSPA $CODE$

	.align 4
	.EXPORT main,CODE			; procedure main defined
	.EXPORT main,ENTRY,PRIV_LEV=3,RTNVAL=GR
main
	.PROC
	.CALLINFO FRAME=128,CALLS,SAVE_RP,SAVE_SP,ENTRY_GR=3
	.ENTRY
	stw %r2,-20(0,%r30)
	copy %r3,%r1
	copy %r30,%r3
	stwm %r1,128(0,%r30)
	.CALL
	bl __main,%r2				; call main procedure
	nop
	addil L'p-$global$,%r27
	ldo R'p-$global$(%r1),%r19
	addil L'a-$global$,%r27
	ldo R'a-$global$(%r1),%r20
```

```
        stw %r20,0(0,%r19)                      ; p = a
        stw 0,8(0,%r3)                          ; k = 0
L$0002                                          ; start loop
        ldw 8(0,%r3),%r19
        ldi 99,%r20
        comclr,>= %r20,%r19,0                   ; check loop condition
        bl,n L$0003,0                           ; exit loop upon condition
        ldi 4,%r26
        .CALL ARGW0=GR
        bl malloc,%r2                           ; call malloc
        nop
        copy %r28,%r19
        addil L'p-$global$,%r27
        ldo R'p-$global$(%r1),%r21
        ldw 0(0,%r21),%r20
        stw %r19,0(0,%r20)                      ; p → q = malloc'd space
        addil L'i-$global$,%r27
        ldo R'i-$global$(%r1),%r19
        addil L'i-$global$,%r27
        ldo R'i-$global$(%r1),%r20
        ldw 8(0,%r3),%r21
        ldw 0(0,%r20),%r20
        add %r21,%r20,%r21
        stw %r21,0(0,%r19)                      ; i = k + i
        addil L'p-$global$,%r27
        ldo R'p-$global$(%r1),%r19
        ldw 0(0,%r19),%r20
        ldw 0(0,%r20),%r19
        addil L'i-$global$,%r27
        ldo R'i-$global$(%r1),%r20
        ldw 0(0,%r20),%r21
        stw %r21,0(0,%r19)                      ; p → q → r = i
        addil L'p-$global$,%r27
        ldo R'p-$global$(%r1),%r19
        addil L'p-$global$,%r27
        ldo R'p-$global$(%r1),%r19
        addil L'p-$global$,%r27
        ldo R'p-$global$(%r1),%r20
        ldw 0(0,%r20),%r21
        ldo 4(%r21),%r20
        copy %r20,%r21
        stw %r21,0(0,%r19)                      ; p++
L$0004
        ldw 8(0,%r3),%r20
        ldo 1(%r20),%r19
```

```
        copy %r19,%r20
        stw %r20,8(0,%r3)                    ; k++
        bl,n L$0002,0                        ; go back to top of loop
L$0003                                       ; end of loop
        addil L'result-$global$,%r27
        ldo R'result-$global$(%r1),%r19
        addil L'a-$global$,%r27
        ldo R'a-$global$(%r1),%r20
        ldo 0(%r20),%r21
        ldw 396(0,%r21),%r20
        ldw 0(0,%r20),%r21
        stw %r21,0(0,%r19)                   ; result = a[99].q -> r
L$0001
        ldw -20(0,%r3),%r2
        ldo 64(%r3),%r30
        ldwm -64(0,%r30),%r3
        bv,n 0(%r2)                          ; end of program
        .EXIT
        .PROCEND
        .SPACE $PRIVATE$
        .SUBSPA $BSS$

p       .comm 8                              ; variable p
result  .comm 8                              ; variable result
```

APPENDIX B

SURVEY RESULTS

B.1 Survey results

Table B.1 shows the median response of the small informal survey conducted to get an indication of how watchpoints are used.

Table B.1. Survey results. On a scale from 0 to 5 how often are watchpoints used.

single global	4
multiple global	2
single heap	3
multiple heap	2
whole array	1
array element	3
whole data structure	2
element of data structure	3
complicated expression	1
local/stack	2

APPENDIX C

GLOBAL TEST RESULTS

C.1 Global test results

Tables C.1, C.2, C.3 show the median, maximum and minimum times of the global tests for each of the programs in the test suite.

Tables C.4, C.5, C.6 show the median, maximum and minimum times of the global tests with the work-arounds the programs in the test suite that performed poorly otherwise. Also the factors by which the overhead is improved are shown in the columns labeled OH Impr.

Tables C.7, C.8, C.9, C.10 show the histograms of the running times of the global tests for each of the programs in the test suite. Each column represents the number of tests that finished in the given range of time in seconds.

Table C.1. Median time for global tests in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57.3	2750000	69.6	1700	269	81.2	65.5
esp	30.4	1370000	55.7	9450	2420	159	75.7
xli	9.3	458000	20.3	12700	3360	190	71.2
gcc	74.4	3440000	126	643	234	83.2	77.7

Table C.2. Maximum time for global tests in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57.3	2750000	4060	2780	559	138	122
esp	30.4	1370000	980000	63500	6820	5800	5970
xli	9.3	458000	206000	19800	5430	2930	3030
gcc	74.4	3440000	87000	182000	46700	1950	2150

Table C.3. Minimum time for global tests in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57.3	2750000	59.1	58.5	57.7	57.4	57.4
esp	30.4	1370000	44.9	35.2	33.3	30.5	30.4
xli	9.3	458000	20.1	16.3	11.1	9.4	9.3
gcc	74.4	3440000	125	75.9	75.0	74.3	74.2

Table C.4. Median time for work-around global tests in seconds.

	no wp	User VM	OH Impr	User B-bit	OH Impr	Krn VM	OH Impr	Krn B-bit	OH Impr
eqn	57.3	65.5	200	59.4	100	57.5	100	57.5	40
esp	30.4	33.6	3000	30.9	5000	30.6	600	30.6	200
xli	9.3	57.5	300	21.4	300	9.9	300	9.5	300

Table C.5. Maximum time for work-around global tests in seconds.

	no wp	User VM	OH Impr	User B-bit	OH Impr	Krn VM	OH Impr	Krn B-bit	OH Impr
eqn	57.3	2780	1	269	2.5	112	1.5	115	1.1
esp	30.4	10900	6	6800	1	5900	1	5780	1
xli	9.3	13800	1.4	5750	1	2850	1	2940	1

Table C.6. Minimum time for work-around global tests in seconds.

	no wp	User VM	OH Impr	User B-bit	OH Impr	Krn VM	OH Impr	Krn B-bit	OH Impr
eqn	57.3	58.5	1	57.8	1	57.4	1	57.3	1
esp	30.4	30.8	12	30.6	14	30.5	1	30.4	1
xli	9.3	13.9	1.5	11.2	1	9.3	1	9.2	1

Table C.7. Histogram of running times for eqntott global tests in seconds.

	—65	65—85	85—125	125–500	500—1000	1000—
Code Patch	31	5	5	11	0	1
User VM	13	6	4	1	2	27
User B-bit	20	3	1	4	25	0
Krn VM	23	28	1	1	0	0
Krn B-bit	23	28	2	0	0	0
User VM Work	13	31	4	2	2	1
User B-bit Work	45	3	2	3	0	0
Krn VM Work	48	3	2	0	0	0
Krn B-bit Work	48	3	2	0	0	0

Table C.8. Histogram of running times for espresso global tests in seconds.

	—35	35—65	65—100	100–500	500—1000	1000—
Code Patch	0	52	0	1	0	2
User VM	0	4	0	4	3	44
User B-bit	4	5	1	1	0	42
Krn VM	9	1	0	44	0	1
Krn B-bit	9	1	44	0	0	1
User VM Work	47	3	4	0	0	1
User B-bit Work	51	3	0	0	0	1
Krn VM Work	53	1	0	0	0	1
Krn B-bit Work	53	1	0	0	0	1

Table C.9. Histogram of running times for xlisp global tests in seconds.

	—20	20—60	60—100	100–500	500—1000	1000—
Code Patch	0	66	1	2	0	8
User VM	1	0	0	0	0	76
User B-bit	1	0	0	0	0	76
Krn VM	1	0	0	71	0	5
Krn B-bit	1	0	71	0	0	5
User VM Work	9	61	1	0	0	6
User B-bit Work	9	62	0	0	0	6
Krn VM Work	69	2	0	1	0	5
Krn B-bit Work	69	2	0	1	0	5

Table C.10. Histogram of running times for gcc global tests in seconds.

	—65	65—85	85—125	125–500	500—1000	1000—
Code Patch	0	0	690	56	20	46
User VM	14	18	134	162	194	290
User B-bit	26	92	218	286	67	123
Krn VM	299	293	97	113	8	2
Krn B-bit	480	202	103	24	3	0

APPENDIX D

HEAP TEST RESULTS

D.1 Heap test results

Tables D.1, D.2, D.3 show the median, maximum and minimum times of the heap tests for each of the programs in the test suite.

Table D.4 shows the median times of the heap tests with the work-arounds for the programs in the test suite that performed poorly otherwise. Also the factors by which the overhead is improved are shown in the columns labeled OH Impr.

Tables D.5, D.6, D.7, D.8 show the histograms of the running times of the heap tests for each of the programs in the test suite. Each column represents the number of tests that finished in the given range of time in seconds.

Table D.1. Median for time heap tests in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57.3	2750000	77.1	76.7	61.8	57.8	57.9
esp	30.4	1370000	55.9	17400	2450	160	74.3
xli	9.3	458000	641	16300	3690	212	83.2
gcc	74.4	3440000	126	767	215.1	81.1	76.7

Table D.2. Maximum time for heap tests in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57.3	2750000	119	1060	269	63.1	61.6
xli	9.3	458000	643	17800	3990	223	86.5
gcc	74.4	3440000	87000	205000	50200	6430	2230

Table D.3. Minimum time for heap tests in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57.3	2750000	70.8	65.5	59.4	57.8	57.6
xli	9.3	458000	20.7	1810	414	37	22.9
gcc	74.4	3440000	126	76.9	75.5	74.6	74.3

Table D.4. Median time for work-around heap tests in seconds.

	no wp	User VM	OH Impr	User B-bit	OH Impr	Krn VM	OH Impr	Krn B-bit	OH Impr
esp	30.4	33	6000	31	2000	31	100	30	100
xli	9.3	1804	10	392	10	35.4	10	21.3	10

Table D.5. Histogram of running times for eqntott heap tests in seconds.

	—65	65—85	85—125	125–500	500—1000	1000—
Code Patch	0	2	1	0	0	0
User VM	2	0	0	0	0	1
User B-bit	2	0	0	1	0	0
Krn VM	3	0	0	0	0	0
Krn B-bit	3	0	0	0	0	0

Table D.6. Histogram of running times for espresso heap tests in seconds.

	–35	35—65	65—100	100–500	500—1000	1000–
Code Patch	0	1	0	0	0	0
User VM	0	0	0	0	0	1
User B-bit	0	0	0	0	0	1
Krn VM	0	0	0	1	0	0
Krn B-bit	0	0	1	0	0	0
User VM Work	1	0	0	0	0	0
User B-bit Work	1	0	0	0	0	0
Krn VM Work	1	0	0	0	0	0
Krn B-bit Work	1	0	0	0	0	0

Table D.7. Histogram of running times for xlisp heap tests in seconds.

	—60	60—100	100–500	500—1000	1000—10000	10000—
Code Patch	3	0	8	45	0	0
User VM	0	0	0	0	3	53
User B-bit	0	0	1	0	55	0
Krn VM	1	0	55	0	0	0
Krn B-bit	2	54	0	0	0	0
User VM Work	0	0	0	1	55	0
User B-bit Work	0	0	53	3	0	0
Krn VM Work	56	0	0	0	0	0
Krn B-bit Work	56	0	0	0	0	0

Table D.8. Histogram of running times for gcc heap tests in seconds.

	—65	65—85	85—125	125–500	500—1000	1000—
Code Patch	0	0	200	6	1	10
User VM	3	6	29	40	53	86
User B-bit	8	25	60	85	29	10
Krn VM	96	82	29	3	1	6
Krn B-bit	156	51	3	0	5	2

APPENDIX E

STACK TEST RESULTS

E.1 Stack test results

Table E.1 shows the results of the stack test for each of the programs in the test suite.

Table E.1. Results of stack test in seconds.

	no wp	single step	code patch	User VM	User B-bit	Krn VM	Krn B-bit
eqn	57	2750000	70	98100	23400	2080	431
esp	30	1370000	54	87700	20500	1120	382
xli	9	458000	21	3000	559	40	18
gcc	74	3440000	129	3610	840	296	87

REFERENCES

[1] Bert Beander. Vax debug: an interactive, symbolic, multilingual debugger. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 173–179, August 1983. Appeared as SIGPLAN Notices 18(8).

[2] Robert Bedichek. A multi-space operating system debugger. University of Washington, Unpublished.

[3] Robert Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the 1990 Winter Usenix Conference*, pages 53–63, October 1989.

[4] J. Brown and R.Klamann. The application of code instrumentation technology in the Los Alamos Debugger. In *Proceedings of the Supercomputer Debugging Workshop*, October 1992.

[5] Steve Burling. Personal communication. MTS developer at University of Michigan.

[6] Peter Chang. Personal communication. Stanford University.

[7] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report (in preparation), Sun Microsystems Laboratories and University of Washington, 1993.

[8] Hewlett Packard Company. *Precision Architecture and Instruction Set Manual.* Manual Part Number 09740-90014, third edition, April 1989.

[9] Hewlett Packard Company. *HP-UX Reference Guide—Vol 2: Section 2.* HP Part Number B2355-90033, E0892 edition, August 1992.

[10] Hewlett Packard Company. *HP-UX Symbolic Debugger User's Guide.* HP Part Number B2355-90044, first edition, August 1992.

[11] Max Copperman and Jeff Thomas. Poor man's watchpoints. Xerox Palo Alto Research Center and Kubota Pacific Computer, Inc, Unpublished.

[12] Intel Corporation. *Intel 80386 Programmer's Reference Manual.* Santa Clara, California, 1986.

[13] Sequent Computer Corporation. *Dynix/ptx Unix System V.4 debug manual page.*

[14] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 Usenix Winter Conference*, pages 125–136, January 1992.

[15] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, New Jersy, 1992.

[16] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the 1988 Usenix Summer Conference*, pages 161–171, June 1988.

[17] David Keppel. Fast data breakpoints. Technical Report 93-04-06, University of Washington, 1993.

[18] Peter Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 78–84, June 1990. Appeared as SIGPLAN Notices 25(6).

[19] Sun Microsystems Laboratories. *SunOS Release 4.1 dbx manual page*, 1989.

[20] Hitatchi Computer Products. *OSF/1 Unix sdb manual page*.

[21] Steven K Reinhardt, Mark D. Hill, Alvin R. Lebeck James R Larus, James C. Lewis, and David A. Wood. Virtual prototyping of parallel computers. In *ACM SIGMETRICS*, May 1993.

[22] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 545 Tech Square, Cambridge, Ma 02139, 4.12 edition, January 1994.

[23] S. Venkata Subramonyam. Personal communication. Code developer for Sequent Computer Corporation.

[24] Robert Wahbe. Efficient data breakpoints. In *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 200–212, October 1992.

[25] Robert Wahbe, Steven Lucco, and Susan L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the ACMSIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 1–12, June 1993.

[26] Paul R. Wilson and Thomas G. Moher. Demonic memories for process histories. In *Proceedings of the ACM '89 Conference on Programming Language Design and Implementation*, pages 330–343, June 1989.