Omnipresent and low-overhead application debugging

Robert Strandh
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The state of the art in application debugging in free Common Lisp implementations leaves much to be desired. In many cases, only a backtrace inspector is provided, allowing the application programmer to examine the control stack when an unhandled error is signaled. Most such implementations do not allow the programmer to set breakpoints (unconditional or conditional), nor to step the program after it has stopped.

Furthermore, even debugging tools such as tracing or manually calling `break` are typically very limited in that they do not allow the programmer to trace or break in important system functions such as `make-instance` or `shared-initialize`, simply because these tools impact all callers, including those of the system itself, such as the compiler.

In this paper, we suggest a technique that solves most of these problems. The main idea is to have a *debugger thread* debug one or more *application threads*, with all these threads running in the same image. Tracing and setting breakpoints have an effect only in the debugged thread so that system code running in other threads is not impacted. We discuss several advantages of this technique, and in particular how it can make debugging *omnipresent*, i.e., not requiring recompilation to get its benefits. We describe how to achieve this advantage while keeping the *overhead* low for threads that are not being debugged.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Runtime environments**;

KEYWORDS

CLOS, Common Lisp, Compilation, Debugging

ACM Reference Format:
Robert Strandh. 2021. Omnipresent and low-overhead application debugging. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3747548

1 INTRODUCTION

Good debugging tools are essential for the productivity of software developers. In this paper, we are concerned with *application programmers* as opposed to *system programmers*. The difference, in the context of this paper, is that the techniques that we suggest are not adapted to debugging the system itself, such as the compiler. Instead, throughout this paper, we assume that, as far as the application programmer is concerned, the semantics of the code generated by the compiler corresponds to that of the source code.

In this paper, we are mainly concerned with Common Lisp [1] implementations distributed as so-called FLOSS, i.e., “Free, Libre, and Open Source Software”. While some such implementations are excellent in terms of the quality of the code that the compiler generates, most leave much to be desired when it comes to debugging tools available to the application programmer.

Perhaps the most advanced development environment available to application programmers using FLOSS Common Lisp implementations is the one that consists of GNU Emacs[1] (also [4] [6]) with SLIME[2]. Many application programmers consider this development environment to be outstanding. Some even believe that it is one of the best, no matter the programming language under consideration.

However, although this environment does a fairly good job with exploiting the features of the Common Lisp implementations that it supports, limitations of those implementations severely restrict what the application programmer can do. In particular, most of these implementations have only very limited facilities for setting breakpoints (unconditional or conditional) and for stepping.

Even in implementations that allow the programmer to set a breakpoint in some code, the places where it is allowed are necessarily restricted, given how breakpoints are typically implemented. The reason for this restriction is that such a breakpoint would be visible to all callers of the code in which the breakpoint is set. When these callers include important system code such as the compiler, or perhaps the debugger itself, setting such a breakpoint would make the entire system useless. This restriction typically applies also to tracing. Most Common Lisp implementations would either not allow for the programmer to trace important system functions such as `make-instance` or `shared-initialize`, or these functions would be rendered useless with any such attempt. The reason is of course that these functions would be called by the system itself, so that output would be drowned in traces of calls that are unimportant to the application programmer.

In this paper, we suggest a technique that solves these problems. The key features of this technique is that breakpoints and traces take effect only in a thread that is executed

ELS'20, April 27–28 2020, Zürich, Switzerland

https://doi.org/10.5281/zenodo.3747548

[1] https://www.gnu.org/software/emacs/manual/emacs.html
[2] https://common-lisp.net/project/slime/doc/html/

from a special *debugger thread*. Thus, even though a function might contain a breakpoint, when that function is called as a normal part of an application, the breakpoint will not have any effect. Only when that function is called (directly or indirectly) from the special debugger thread is the breakpoint visible.

The technique presented in this paper is yet to be implemented. We have, however, conducted experiments that suggest that it is entirely viable. We plan to make it the default technique used in our system SICL (see Section 3), currently under development.

Throughout this paper, we use the term *user* to mean the person operating the debugger or some debugging-related facility, so as to distinguish this person from the *application programmer*, by which we mean the author of the code being debugged. The two can obviously be one and the same person in two different roles.

2 PREVIOUS WORK

2.1 Process-based debugging

With systems like UNIX, debugging is usually performed as an interaction between two *processes*. The debugger runs in one process and the application in another process. For a breakpoint, the code of the application is modified by the debugger so that the application sends a signal to the debugger when the breakpoint has been reached. For this purpose, the debugger maps the code pages of the application as *copy on write* (or COW). With this technique, instances of the same application that are not executed under the control of the debugger are not affected by the modified code. In particular, with this technique, any application can be debugged, including the debuggger itself.

Some FLOSS Common Lisp implementations suggest the use of this debugging technique, by means of some existing debugger such as GDB[3] (also [7]), in order to set breakpoints. In particular, the CCL (See Section 2.3.) documentation mentions that this technique is possible, and it is also the technique recommended for ECL (See Section 2.4.).

2.2 SBCL

The SBCL Common Lisp implementation[4] has a breakpoint facility. Given a code location, a breakpoint can be set, which results in the code being modified at that location, so that an arbitrary function (given to the constructor of the breakpoint) is called when execution reaches that location.

The only feature that uses the breakpoint facility is `trace`. Furthermore, it is hard for the user to take advantage of the breakpoint facility directly, given that a function such as `make-breakpoint` requires an argument indicating the code location. We are unaware of the existence of a debugger for SBCL that can use the breakpoint facility.

SBCL also has a *single stepper* that the manual says is "instrumentation based". As it turns out, the kind of instrumentation used by the stepper is not that of the breakpoint facility. Instead, when the value of the `debug` optimization quality is sufficiently high compared to the values of other optimization qualities, the compiler inserts code that signals conditions that are specific to the stepper.

2.3 CCL

The CCL Common Lisp implementation[5] does not have the concept of breakpoints.

The CCL `trace` command uses *encapsulation*, meaning that the association between the *name* of a function and the function object itself is altered so that it contains a *wrapper* function that displays the information requested and that calls the original function to accomplish its task.

Currently, CCL does not have a working single stepper.

2.4 ECL

The ECL Common Lisp implementation[6] does not have the concept of breakpoints, so an external debugger such as GDB has to be used for breakpoints. ECL does have a special instruction type in the bytecode virtual machine that is used for stepping.

The `trace` facility uses encapsulation.

2.5 Clasp

The Clasp Common Lisp implementation[7] does not have the concept of breakpoints, nor does it have a stepper. The `trace` facility uses encapsulation.

2.6 LispWorks

The LispWorks Common Lisp implementation[8] provides breakpoints. Breakpoints can be set either from the stepper or from the editor. The first time a breakpoint is set in a definition, the source code of the defining form is re-evaluated with additional annotations that provide information for the stepper.

When a breakpoint has been set, it is active no matter how the code containing it was called. If that code was called outside the stepper, the stepper is automatically started. Thus, breakpoints provide the essential mechanism for the stepper.

Since setting a breakpoint requires access to the source code, and since the source code of the system itself is not supplied, the user can not set breakpoints in system code.

The `trace` facility in LispWorks is accomplished through encapsulation.

2.7 Allegro

The Allegro Common Lisp implementation[9] has the most complete and most sophisticated implementation of breakpoints of all the Common Lisp implementations we investigated.

[3] https://sourceware.org/gdb/current/onlinedocs/gdb/

[4] http://www.sbcl.org/

[5] https://ccl.clozure.com/

[6] https://common-lisp.net/project/ecl/

[7] https://github.com/clasp-developers/clasp

[8] http://www.lispworks.com/products/lispworks.html

[9] https://franz.com/products/allegro-common-lisp/

High-level debugging features are based on a low-level breakpoint mechanism described in a paper by Duane Rettig [5]. In many respects, the low-level mechanism is similar to the one used by UNIX-style debugging, in that it replaces the ordinary machine instruction by one that will *trap*, and thus cause the operating system to send a signal to the Lisp process. However, their mechanism differs in a significant way from the one used by UNIX-style debugging, in that it allows the breakpoint to be handled by the same operating-system process that contains it, with very few exceptions.

Same-process debugging is made possible by their mechanism that allows existing breakpoints to be *installed* or not. Only installed breakpoints correspond to replaced instructions, whereas uninstalled breakpoints are remembered by the system and can be installed according to the kind of debugging that the higher-level tool implements. The clever aspect of their mechanism is to have the signal handler start its action by uninstalling all breakpoints. Thus, even if a breakpoint exists in some system code that is also used by the debugger, once the debugger is entered, the breakpoint is no longer active. Had the breakpoints remained installed, issuing commands inside the debugger might have invoked some code with a breakpoint, thereby halting the execution of the debugger itself.

This mechanism allows for instruction-level stepping in a way similar to what is possible in separate-process UNIX-style debuggers. Just as with UNIX-style debugging, any instruction can be replaced by a different one that will trap to the debugger. As a result, it is possible to execute one instruction at a time by simply trapping after each instruction. Crucially, however, this mechanism is then used to build high-level tools such as source-level debuggers, steppers, etc.

3 MAIN FEATURES OF THE SICL SYSTEM

SICL[10] is a system that is written entirely in Common Lisp. Thanks to the particular bootstrapping technique [2] that we developed for SICL, most parts of the system can use the entire language for their implementation. We thus avoid having to keep track of what particular subset of the language is allowed for the implementation of each module.

We have multiple objectives for the SICL system, including exemplary maintainability and good performance. However, the most important objective in the context of this paper is *support for excellent debugging tools*. We think it is going to be difficult to adapt existing Common Lisp implementations to support the kind of application debugging that we consider essential for good programmer productivity.

Another main objective of the SICL system is *safety*. In this context, by this term we mean that the system must always be in a coherent internal state. When a system becomes unsafe, it may *crash*, or (worse) silently produce the wrong answer.

There are many situations described in the Common Lisp standard that have undefined or unspecified behavior, such as:

(1) Many times when a standard function is called with some argument that is not of the type indicated by the corresponding dictionary entry in the Common Lisp standard document, the behavior is undefined, allowing the implementation to avoid potentially costly tests for exceptional situations.
(2) When a non-local transfer is attempted to an exit point that has been "abandoned", the standard does not require this situation to be detected, making it possible for the system to crash or (worse) give the wrong result.
(3) When some entity is declared `dynamic-extent`, but some necessary condition for this declaration is violated, the implementation is again not required to detect the problem, again potentially resulting in a crash or an incorrect computation.

Fortunately, most potential situations of this type are not taken advantage of by a typical Common Lisp implementation in order to improve performance, but some are. We think that the spirit of the Common Lisp standard is to have a safe language, and that many of these situations of undefined or unspecified behavior exist only to avoid significantly more work for the system maintainers at the time the standard was established.

For that reason, in the SICL system, we do not intend to take advantage of these situations to make the system unsafe for the purpose of better performance, even though we might have to work somewhat harder in order to maintain good performance in all situations.

Many debugging techniques can make the system unsafe. For example, if the debugger allows the user to arbitrarily change the value of a lexical variable, the new value might violate some assumption made by the compiler for the program point in question. Such a violation is very likely to make the system unsafe. The work described in this paper is designed to keep the system safe.

4 OUR TECHNIQUE

4.1 Two versions of every function body

We provide two different versions of every function body[11]. One version, called the *ordinary* version, and the other one is called the *debugging* version. Each version is provided as a separate *entry point* for the function[12]. The two versions are *similar* (but not identical) copies of the entire function body.

By including both versions in the same function, we make it unnecessary for the application programmer to recompile the code with higher debug settings when it is desirable to have more debugging information than what the compiler would generate by default.

[10] https://github.com/robert-strandh/SICL

[11] This idea was suggested by Michael Raskin.

[12] This idea was suggested by Frode Fjeld.

The ordinary function body is compiled using every typical optimization technique used by a good compiler, including:

- constant folding,
- dead code elimination,
- common sub-expression elimination,
- loop-invariant code motion,
- induction-variable optimization,
- elimination of in-scope dead variables, and
- tail-call optimization.

Some of these optimization techniques are essential for high-performance code, but many of them can make it significantly harder for the user to understand what the program is doing:

- Common sub-expression elimination and similar techniques for redundancy elimination may make it impossible to set a breakpoint in some part of the code, simply because that code has been eliminated by the compiler.
- When a variable is used for the last time, the compiler typically reuses the place that it occupies for other purposes, even though the variable may still be in scope. This optimization makes it impossible for the user to examine the value of a variable that has been eliminated. A user with a poor understanding of compiler-optimization techniques will find the result surprising.
- Loop-invariant code motion results in code being moved from inside a loop to outside it. Any attempt by the application programmer to set a breakpoint in such code will fail.
- Induction-variable optimization will eliminate or replace variables in source code by others that are more beneficial for the performance of the computation, again making it harder for the user to debug the code.
- Tail-call optimization exploits the fact that the stack needs to reflect only the *future* of the computation, but the *past* can be omitted. For the purpose of debugging, the past of the computation provides essential information to the developer.

To avoid many of these inconveniences to the user, the debugging version of the function body is compiled in a way that makes the code somewhat slower, but much more friendly for the purpose of debugging. Some of the optimization techniques cited above will not be performed at all, or only in a less "aggressive" form. Messages from the compiler (such as when dead code is eliminated) are emitted based on the compilation of the normal version of the function body so that the maximum amount feedback is obtained.

Notice that the existence of the two versions of the function body makes it usually unnecessary for the programmer to explicitly indicate values of the `debug` optimize quality. In fact, forcing the programmer to set this value is often a major inconvenience, since it typically requires the programmer to choose between debugging convenience and execution speed. The `speed` and `compilation-speed` optimize qualities may influence compilation of the the normal version of the body, but not the debugging version. The debugging version will in general be much faster to generate because of the absence of most optimization passes.

Furthermore, the debugging version of the code is compiled so that a small routine is called immediately before and immediately after the execution corresponding to the evaluation of a form in the source code. In order to determine whether a breakpoint is present at that particular location in the source code, this routine performs a query of a table managed by the debugger. While the details of how this table is implemented might evolve, here we give one example of such an implementation, thereby arguing that performing a query is not overly expensive in terms of performance.

As an example of implementation of this table, it might be split into two sub-tables called the *summary table* and the *breakpoint table*. Both these tables are managed by the debugger, in that actions on the part of the user may alter their contents. The application consults these tables, directly or indirectly, to determine whether a breakpoint is present.

The purpose of the summary table is to provide a quick test that almost always indicates that no breakpoint is present. Thus, the summary table is a fixed-size bit vector. The size will typically be a small power of 2, for instance 1024 which represents a modest 16 64-bit words on a modern processor. The application routine computes the value of the program counter modulo the size of the table in order to determine an index. If the entry in the table contains 0, then there is definitely not a breakpoint present at the source location in question. Since there are typically only a modest number of breakpoints in a program, most of the time, the entry will contain a 0, making the routine return immediately, and normal form evaluation to continue. The debugging version of the function body accesses this table early on in order to create a reference to it in a lexical variable. This lexical variable is subject to register allocation as usual.

If the entry in the summary table contains 1, then there is a breakpoint at *some* value of the program counter that, when taken modulo the size of the table, has a breakpoint present. If this is the case, then the routine consults the breakpoint table. In other words, the summary table acts as a Bloom filter [3], in that false positives are possible, but false negatives are not. The size of the table determines the probability of a false positive.

The breakpoint table is a hash table in which the keys are values of the program counter[13] and the values are objects that the debugger uses in order to determine information about the breakpoint in question. When the routine finds that a breakpoint is present at the current source location, it informs the debugger. Details of the communication between the application thread and the debugger are discussed in Section 4.2.

In the ordinary version of the function body, when a function call is made, the caller uses the entry point of the callee corresponding to the ordinary version of the body of the callee. In the debugging version of the function body, on the

[13] In implementations where code can be moved by the garbage collector, this table must be re-hashed after a collection. The tentative decision for SICL is to have all code at fixed locations.

other hand, when a function call is made, the caller uses the entry point of the callee corresponding to the debugging version of the body of the callee. This mechanism thus automatically propagates the information about debugging throughout the call chain.

4.2 Communication between the debugger and the application

Debuggers in UNIX systems have full access to the address space of the application, including the stack and the lexical variables. A UNIX debugger can therefore modify any lexical variable and then continue the execution of the application. Such manipulations may very well violate some of the assumptions made by the compiler for a particular code fragment. For example, if the code contains a test for the value of a numeric variable, the compiler may make different assumptions about this value in the two different branches executed as a result of the test.

Allowing a debugger to make arbitrary modifications to lexical variables, let alone to any memory location, in a Common Lisp application program will defeat any attempts at making the system safe, and safety is one of the objectives of the SICL system as expressed in Section 3. We must therefore come up with a different communication protocol that keeps the system safe.

Our design contains two essential elements for this purpose:

(1) The debugger consists of an interactive application with a *command loop*. An iteration of this command loop can of course be prompted by a user interaction. However, when the application detects a breakpoint by querying the tables described in Section 4.1, it injects a command into the command loop of the debugger, triggering the execution of code in the debugger to handle the breakpoint.

(2) A shared *queue* is used to send messages from the debugger to the application. This queue has a *semaphore* associated with it.

(3) Once the application has informed the debugger about a breakpoint, it attempts to *dequeue* the next message on the queue. If the queue is empty, the application automatically *waits* on the associated semaphore, until the debugger issues an *enqueue* operation with instructions for the application.

The debugger is in charge of taking into account the commands issued by the user. When the user indicates that a certain action should be performed at a particular place in the source code, the debugger populates the two tables mentioned in Section 4.1, and records the particular action the user desires, for example:

- The user might indicate that the application should stop and wait for further actions by the user, after the user has examined the state of application data. In this case, the debugger records this desire in the breakpoint table. When the application then reaches the breakpoint in question, the debugger waits for further user action in its command loop.

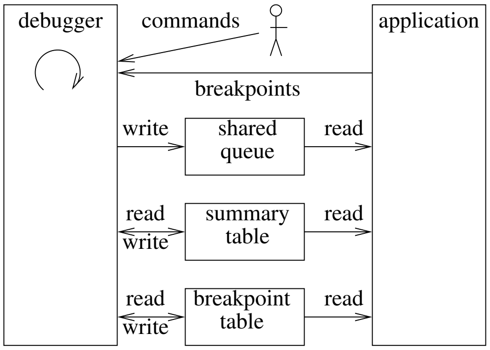

Figure 1: Communication between user, application, and debugger.

- After the user has examined the state of application data as a result of the application having stopped, the user can issue a command that makes the application continue normal execution. The debugger then immediately sends a message to this effect to the application.
- The user might indicate that a *trace message* should be printed without stopping the application. Then, when the breakpoint is reached, the debugger displays the message to the user and sends a message to the application to continue execution.
- The user can also indicate that the execution of the application should be *stepped* in one of several different ways:
  - *next*: Execution stops at the next possible program point.
  - *in*: Execution stops at the beginning of a function being called.
  - *out*: The remaining sub-forms of the form containing the current breakpoint are evaluated, and execution stops immediately after the evaluation of that form.
  - *over*: When a breakpoint is reached that is located immediately before a form is evaluated, the form is evaluated and execution stops immediately after this evaluation.
  - *finish*: Execution of the currently executing function terminates, and stops in the calling function immediately after the call.

  The debugger then sets one or more *volatile breakpoints* (i.e., breakpoints that will be removed once reached) at source locations corresponding to the type of stepping required. It then instructs the application to continue execution as usual.

To allow for the user to examine the state of the application, when the application thread detects a breakpoint, the command it injects into the debugger command loop contains a complete list of live local variables and their values, as well as of special variables bound in the application thread.

Since we intend to provide debugger commands for examining and modifying application data, we must make sure that any such manipulation on the part of the user preserves the integrity of the application.

In particular, any assumptions made by the compiler about the structure or type of some lexical variable must be impossible to violate through the modification of the value of a lexical variable. We obtain this property by making sure that the compiler does not propagate any information about the structure or type of lexical variables between program points that admit breakpoints. Thus, any run-time manipulation that requires this structure or type to be known must be preceded by an explicit test, and the compiler does not generate code that admits a breakpoint between the test and the manipulation.

4.3 Debugger commands available to the user

We have an embryonic implementation of an interactive debugger, called Clordane.[14] We use the McCLIM library for writing graphic user interfaces as a basis for this tool. Currently, Clordane can show the source code of an application (one source file at a time) and an indication of the place of a breakpoint. The application being debugged is then compiled with the SICL compiler, generating high-level intermediate representation (HIR). The HIR code is then interpreted by a small program running in a host Common Lisp implementation.

The communication protocol described in Section 4.2 has been implemented and works to our satisfaction, but only a small subset of interactions have been implemented so far.

We think that the following commands must be implemented in a fully featured debugger:

- The user should be able to point to a location in the source code to indicate a particular action to be taken at that point:
  - Stop the execution of the application and wait for further actions.
  - Print a trace message, possibly containing the values of live variables, and then continue the execution.

  It should be possible to make the action conditional, based on some arbitrary expression to be evaluated in the debugger thread. This expression can contain references to live variables in the application.
- When the application is stopped, the user should be able to examine live variables, and (in some cases, with restrictions) modify their values.
- Also, when the application is stopped, the application programmer should be able to issue one of several types of *stepping* commands, implicitly indicating the next location for the application to stop.

5 BENEFITS OF OUR TECHNIQUE

Our technique differs both from the tradition of debugging in UNIX-type systems and from the tradition used in FLOSS Common Lisp systems.

5.1 Difference compared to UNIX-like systems

Whereas UNIX-like systems typically run the debugger in a different *process* from that (or those) of the application, with our technique we run both the debugger and the application in the same process.

The main advantage of this organization is that communication between the debugger and the application is greatly simplified. There is no need for a wire protocol to encode and decode data in the form of sequences of octets, simply because with a single process, the address space is shared between the debugger and the application. Instead, we can send data in the form of arbitrarily complex data structures between the two.

5.2 Difference compared to most FLOSS Common Lisp systems

Most FLOSS Common Lisp implementations have a history that started before multi-threading was common. As a result, features such as breakpoints and tracing are often implemented as modifications to the code.

For example, in SBCL, the user can choose to *trace* a function in two different ways. One way is by means of *encapsulation*, meaning that the function is not modified, and instead wrapped in a small routine that then replaces the function as associated with the function *name*. The function being traced is not modified. The other way is by means of a breakpoint; that is, the code of the function being traced *is* modified.

However, in both cases, every caller of the function being traced is affected, barring a caller that is in possession of the function object itself, rather than its name. As a result, it is very likely impractical to trace system functions that may be used internally by the system. For example, tracing `find` or `position` (if at all possible) is likely to generate so much information from callers that are irrelevant to the user as to make the information impossible to exploit. And tracing functions such as `print`, `format`, or `write` would be entirely impossible, since the trace output would very likely call these functions in order to generate the output information meant for the user.

With our suggested technique, tracing a function does not create an encapsulation and does not modify the code of the function. Instead, the existing code communicates with the debugger, and the debugger, running in a different thread, is in charge of displaying information to the user. As a direct consequence, there are no restrictions such as those indicated above. The only possible restriction has to do with inlining, though it may very well turn out to be possible to propagate debugging information with inlining, thereby making it possible to trace, or to set breakpoints in

[14] The name Clordane is a deliberate misspelling of "Chlordane" which is a pesticide that was banned in most countries in the 1980s. The misspelling was designed to suggest the Common Lisp language and to make answers by search engines less cluttered.

any function such as `car` or `+`. However, it may turn out that the inclusion of debugging code in such low-level functions would be prohibitive in terms of performance of code run under the control of the debugger.

Finally, a significant advantage to our technique is that the application programmer does not have to choose between compiling the code for debugging or for performance. In most existing systems, in order for it to be possible to benefit from all the debugging information possible, the programmer has to compile the code with a combination of values of the existing `debug` qualities that is not optimal for performance. This limitation means that it is often necessary to recompile the application for one of the two purposes. With the technique presented in this paper, no such choice is required, since both versions of the application are always available.

6 DISADVANTAGES OF OUR TECHNIQUE

Perhaps the most obvious disadvantage of our technique is that the size of the code will more than double. The debugging version of the function body must implement the same functionality as the non-debugging version, but in addition to that functionality, it must also contain code for communicating with the debugger. Furthermore, since fewer optimizations are applied to the non-debugging version, even without the code for communication, the debugging version would be larger than the non-debugging version.

While the additional code will impact the memory footprint of the system, we do not think it will have any negative influence on caching. The two versions of the body are kept separate, and the same version is typically executed repeatedly.

Feedback on draft versions of this paper indicate that many readers are worried about the possibility of the behavior of the different versions of the function body described in Section 4.1. This worry is based on experience, as this situation is common, especially with implementations of programming languages other than Common Lisp. As we see it, there are two possible causes for such difference in behavior:

(1) A defect in the compiler can result in native code that does not correspond to the semantics of the source code, and the resulting code can be different in the different versions.
(2) The compiler is exploiting undefined or unspecified behavior, probably in order to improve performance of the resulting code, and it exploits such behavior in different ways in the two different versions.

We briefly addressed the first cause in Section 1, by specifically targeting application programming, assuming that the compiler is essentially free of defects.

The second cause was addressed in Section 3, where we indicated that the SICL system does not intend to take advantage of undefined situations that will introduce any such differences in application behavior.

Finally, the fact that the technique proposed in this paper is incompatible with the way most Common Lisp systems work, makes it unlikely that existing systems will be able to use it. We are convinced, however, that our technique will represent a major advantage in terms of productivity for the application programmer.

7 CONCLUSIONS AND FUTURE WORK

We believe that a decent development environment for Common Lisp must include a very feature-full debugger for application programs, and we firmly believe that the best way of accomplishing such an environment is to have the debugger execute in the same process as the application, but to have the application react to debugging operations only when executed under the control of the debugger. Here "debugging operations" include the tracing of function calls, since that mechanism is based on the general breakpoint mechanism described in this paper.

While it may seem like a valid objection that the application programmer, as a result of detecting a defect, must rerun the application from the debugger in order to benefit from the features proposed by our technique, we disagree with this objection. We simply think that there is usually no valid reason to run the application outside the debugger during the development phase. The only exceptions we can think of would be applications with extreme performance requirements, or applications where response time is part of the specification. While our technique requires the programmer to *execute* the application code from within the debugger, *developing* the code can be done using any tool.

The validity of the technique described in this paper has been somewhat verified, in that we have an embryonic implementation of an interactive debugger, and an implementation of a small subset of the communication protocol between the application and the debugger. However, our current environment does not allow us to verify the ultimate performance of application code run under the control of the debugger. We firmly believe that the performance loss for code without any breakpoints in it will be acceptable, and that the additional cost when breakpoints are involved is to be expected by the application programmer.

We have not yet implemented the two-version body idea (See Section 4.1.) in SICL, mainly because the only implementation of SICL that currently works is the interpreter for intermediate code. This interpreter was written for bootstrapping purposes only and performance is not an issue. For that reason, we have included only the debugging version of function bodies.

In addition to producing a native version of the SICL system, we also need additional work on the Clordane debugger, and on the additional components of the communication protocol between the debugger and the application.

We think it would be desirable for existing Common Lisp implementations to incorporate the technique described in this paper, so as to allow for a much more complete development environment for users. However, we are convinced that the modifications that would be required to those implementations would be prohibitive in terms of time invested by maintainers. For that reason, we unfortunately do not hold out much hope for this possibility, and we intend to concentrate our efforts on making our technique work for the SICL system.

8 ACKNOWLEDGMENTS

We would like to thank the following people for providing information about breakpoints, tracing, and stepping in various Common Lisp implementations: Martin Simmons (LispWorks), Michał "phoe" Herda (CCL), Alex Wood (Clasp), Daniel Kochmański (ECL), Duane Rettig (Allegro).

We would like to thank Frode Fjeld for giving feedback on early versions of this paper, and for suggesting the use of multiple entry points for each function.

We would like to thank Michael Raskin for suggesting that two versions of each function should be provided, thereby making it unnecessary for the programmer to choose the level of debugging.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[2] *Bootstrapping Common Lisp using Common Lisp*, April 2019. Zenodo. doi: 10.5281/zenodo.2634314. URL https://doi.org/10.5281/zenodo.2634314.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692. URL https://doi.org/10.1145/362686.362692.

[4] Debra Cameron, James Elliott, Marc Loy, Eric S. Raymond, and Bill Rosenblatt. *Learning Gnu Emacs, Third Edition.* O'Reilly Media, Inc., 3 edition, 2004. ISBN 9780596006488.

[5] D. Rettig. Instruction-Level Breakpoint Stepping in the Current Process. Technical report, Franz Inc., Oakland, California, USA, 1999.

[6] Richard Stallman. *GNU Emacs Manual.* GNU Press, Cambridge, MA, USA, 2018.

[7] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB.* GNU Press, Cambridge, MA, USA, 2020. ISBN 978-0-9831592-3-0.