Call-site optimization for Common Lisp

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

ABSTRACT

A function call in a language such as Common Lisp can be fairly costly. Not only is an indirection required so that a redefinition of the callee can take effect for subsequent calls, but several features of Common Lisp can have an even greater impact on the performance of function calls. The presence of optional parameters and/or keyword parameters requires some non-trivial argument parsing in the callee. And when the callee is a generic function, it must invoke the discriminating function in order to dispatch to the effective method that is determined by the arguments. Restrictions such as the required boxing of all arguments can make function calls slow for full-word integer and floating-point numbers.

In this paper, we propose a very general technique for improving the performance of function calls in Common Lisp. Our technique is based on *call-site optimization*, meaning that each call site can be automatically customized for the callee according to the number and the types of the arguments being transmitted to the callee. Our technique is based on the call site being implemented as an unconditional jump to a *trampoline snippet* that is generated by the callee according to information provided by the caller with respect to the arguments. Thus, the callee is able to fully customize the call, thereby avoiding many costly steps of a function call such as indirections, boxing/unboxing, argument parsing, and more.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; **Runtime environments**;

KEYWORDS

Common Lisp, Performance, Call-site optimization

ACM Reference Format:
Robert Strandh. 2021. Call-site optimization for Common Lisp. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/zenodo.4709958

1 INTRODUCTION

Function calls in a dynamic language like Common Lisp can be significantly more expensive in terms of processor cycles than function calls in a typical static language. There are several reasons for this additional cost:

(1) With *late binding* being a requirement, i.e., the fact that functions can be removed or redefined at run-time, and that callers must take such updates into account, it is necessary to have some indirection that can be modified at run-time. Mechanisms such as compiler macros and inlining break this requirement, which is often a serious drawback to their use.
(2) Common Lisp has a rich function-call protocol with optional parameters and keyword parameters. Keyword parameters, in particular, require some considerable run-time parsing for every call to a function that has such parameters.
(3) In general, a function that can honor its contract only for certain types of its arguments must check such types for each call.
(4) All objects must be *boxed* in order to be used as function arguments. For example, IEEE double-float values will typically have to be allocated on the heap, though so-called NaN-boxing [4] can eliminate that particular case. Full-word integers still require boxing, however. Similarly, boxing is required for values returned by a function.
(5) Generic functions can be dynamically updated by the addition or removal of methods. Thus, even when the callee is a known generic function, callers can make no assumptions about which methods might be applicable.
(6) The fact that a function can return multiple values requires the callee to return additional information about the number of return values, and callers that accept multiple values must retrieve this information in order to access the return values, and use default values when it expects more values than the callee returned.

In a typical Common Lisp implementation, item number 1 is handled by an indirection in the form of a slot in the symbol naming the function, requiring a memory access. On modern processors a memory indirect branch is more costly than a direct branch. Even if the branch-prediction logic of the processor is able to make the right decision in the indirect case, there is at least the additional cost of accessing the cache.

Item number 2 can be mitigated by the use of compiler macros. Essentially, the creator of a function with a non-trivial lambda list can also create special versions of this function for various argument lists. A call with an argument list that is recognized by the compiler macro can then be replaced by a call to such a special version, presumably with a simpler lambda list.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'21, May 03–04 2021, Online, Everywhere
© 2021 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.4709958

Item number 3 can be handled by inlining, allowing the compiler to take advantage of type inference and type declarations to determine that some type checks can be elided. However, inlining has the disadvantage that a redefinition of the callee will not automatically be taken into account, thereby requiring the caller to be recompiled for the redefinition to be effective.

The problem indicated by item number 4 can be largely eliminated by the use of more than one entry point for functions, one of which would accept unboxed arguments. This technique is used by Allegro Common Lisp[1] which also allows for functions to return a single unboxed value.

Concerning item number 5, the main difference between function redefinition and generic-function updates is that a generic function consists of independent *effective methods*, only one of which is applicable for a particular call. To determine which effective method is applicable, in the general case some significant *generic dispatch*, based on the class or the identity of arguments, may be required.

A common technique for handling item number 6 is to recognize that most callers will use only a single return value. Then, if the callee returns no values, the register holding the first return value will nevertheless be initialized to `nil`. As a result, callers that use a single return value never have to test the number of values actually returned.

In this paper, we propose a very general technique for call-site optimization that can handle many of the issues listed at the beginning of this section. We plan to incorporate this technique in the SICL[2] implementation of the Common Lisp language.

2 PREVIOUS WORK

To our knowledge, no work on call-site optimization has been published in the context of Common Lisp, though some practical work exists in the form of code in certain implementations, as explained in Section 2.2.

The absence of published work is perhaps due to the many unique features of Common Lisp functions, that make the task very difficult, such as keyword arguments, generic functions with arbitrary method combinations, etc.

2.1 Inline caching

One technique that is fairly common is *inline caching*, pioneered by Smalltalk [2]. This technique is used to avoid repeated method selection in a particular call site. The key observation is that, for a particular call site, often the same method is concerned each time the call is made. By caching the latest method, keyed by the distinguished class argument, the system can often avoid a costlier computation.

The purpose of inline caching being to reduce the cost of finding the applicable effective method, it is not directly related to speeding up function calls, but it has the effect of making calls to generic functions faster.

[1] https://franz.com/products/allegro-common-lisp/
[2] https://github.com/robert-strandh/SICL

2.2 Ctors

The CMUCL[3] implementation of the Common Lisp language uses a technique that they call *ctors* that can be used for call-site optimization of certain functions. This optimization was introduced by Gerd Moellmann in 2002, and has since been included also in SBCL[4], which is a derivative of CMUCL. In particular, in CMUCL the technique is used for the function `make-instance` which is often called with a literal class name and literal keywords for the initialization arguments. When the name of the class to instantiate is a literal, several steps in the object-initialization protocol can be simplified.

Most importantly, checking the validity of the initialization arguments can be done once and for all, subject only to added or removed methods on the functions `initialize-instance` and `shared-initialize` and to updates to the class being instantiated.

CMUCL accomplishes the optimization by replacing (using a compiler macro) the original call to `make-instance` by a call to a *funcallable object* that is specific to the name of the class and the literal keyword arguments given. The *funcallable instance function* of the funcallable object is updated as a result of added or removed methods and modified classes as mentioned. This technique can be used on other, similar functions. For example, `slot-value` is often called with a constant slot name, and this fact has been explored by SBCL.

Since the optimization is done as a manual source-code transformation, it is applicable mainly to standard functions that can not change later on. The mechanism presented in this paper can be seen as an automatic low-overhead version of ctors.

A similar mechanism (called "constructor functions") exists in Allegro Common Lisp. And the Clasp Common Lisp implementation[5] uses a similar mechanism for `make-instance`, `change-class`, and `reinitialize-instance`.

2.3 Sealing

Sealing is a mechanism that allows the programmer to freeze the definitions of various program elements such as classes and generic functions. The work by Heisig [5] applies to Common Lisp and can allow for certain call sites to be optimized to different degrees, from bypassing generic dispatch to fully inlining entire effective methods.

3 MAIN FEATURES OF THE SICL SYSTEM

In this section, we give a quick overview of the main features of our system SICL. The important aspect of our system in order for the technique described in this paper to work is that code is not moved by the garbage collector, as described below.

SICL is a system that is written entirely in Common Lisp. Thanks to the particular bootstrapping technique [1] that we developed for SICL, most parts of the system can use

[3] https://cmucl.org
[4] http://www.sbcl.org/
[5] https://github.com/clasp-developers/clasp

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 that the design of the garbage collector is such that executable instructions do not move as a result of a collection cycle. Our design is based on that of a concurrent generational collector for the ML language [3]. We use a *nursery* generation for each thread, and a global heap for shared objects. So, for the purpose of the current work, the important feature of the garbage collector is that the objects in the global heap do not move, and that all executable code is allocated in that global heap.

The fact that code does not move is beneficial for the instruction cache; moreover it crucially allows us to allocate different objects in the global heap containing machine instructions, and to use fixed relative addresses to refer to one such object from another such object.

4 OUR TECHNIQUE

4.1 Function call

A function call involves a first-class object called a *function object* or a *function* for short. In general, a function may refer to variables introduced in some outer scope, so that the function is a *closure*. The (typically native) instructions to be executed by the function must be able to refer to such *closed-over* variables. But the values of such variables may vary according to the flow of control at run time. This situation is handled by a compile-time procedure called *closure conversion* whereby a *static environment* is determined for each function object. A function object thus consists at least of an *entry point*, which is the address of the code to be executed (and which is shared between all closures with the same code) and an object representing the static environment (which is specific to each function object). A function call must therefore contain instructions to access the static environment and put it in an agreed-upon place (typically a register), before control is transferred to the entry point.

This work covers function calls to functions that are *named* at the call site. The most common such case is when the name of the function appears in the operator position of a compound form. Less common cases include arguments of the form `(function` *name*`)` to some standard functions such as `funcall` and `apply`. In particular, expansions of the `setf` macro are often of the form `(funcall (function (setf` *symbol*`)) ...)` because function names like `(setf` *symbol*`)` are not allowed in an operator position.

In general, with such named function calls, the function associated with the name can be altered at run time, or it can be made undefined by the use of `fmakunbound`. For that reason, the caller can make no assumptions about the signature of the callee. This issue is solved by a standardized *function-call protocol* that dictates where the caller places the arguments it passes to the callee.

Thus, for the purpose of this work, we define a *function call* to be the code that accomplishes the following tasks:

(1) It accesses the arguments to be passed to the callee from the places they have been stored after computation, and puts the arguments in the places where the callee expects them.
(2) It accesses the function object associated with the name at the call site and stores it in some temporary location.
(3) From the function object, it accesses the static environment to be passed to the code of the callee.
(4) Also from the function object, it accesses the *entry point* of the function, i.e., typically the address of the first instruction of the code of the callee.
(5) It transfers control to the entry point, using an instruction that saves the return address for use by the callee to return to the caller.
(6) Upon callee return, it accesses the return values from the places they have been stored by the callee, and puts those values in the places where the caller requires them for further computation.

In a typical implementation, a function call is generated when the code of the caller is compiled, and it then never changes. As mentioned above, for this permanent code to work, a particular *function-call protocol* must be observed, and that protocol must be independent of the callee, as the callee may change after the caller has been compiled.

Our technique optimizes function calls to functions in the *global environment* such that the *name* of the callee is known statically, i.e., at compile time. There are three different types of forms that correspond to this description and that are considered in this work:

(1) A function form where the operator is a symbol naming a function in the global environment, and that does not correspond to any of the following two form types. We use the term *ordinary function form* for this case.
(2) A function form where the operator is the symbol `funcall` and the first argument is either a literal symbol or a `function` special form with a function name.
(3) A function form where the operator is the symbol `apply` and the first argument is either a literal symbol or a `function` special form with a function name.

The first type of form can be considered as special syntax for a `funcall` form with a constant function name.

There are some other cases that we do not intend to cover, in particular a call to `multiple-value-call` with a named function argument. In fact, the third type of form could be generalized to cover other functions that are commonly used with a constant function argument, such as `mapcar`. At the moment, we are not considering such additional cases.

With our suggested technique, for these three different form types, the function call is created by the callee. We discuss each form type separately.

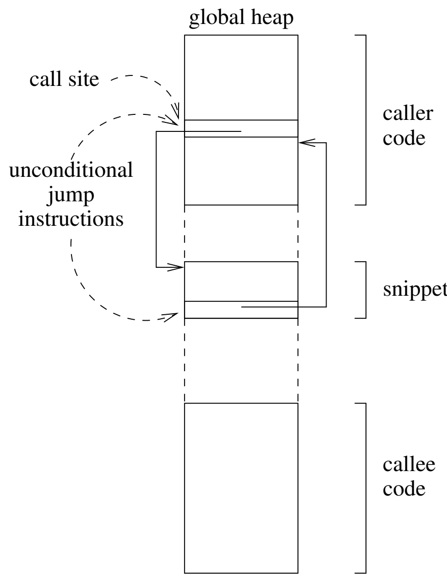

Figure 1: Caller, callee, and snippet in the global heap.

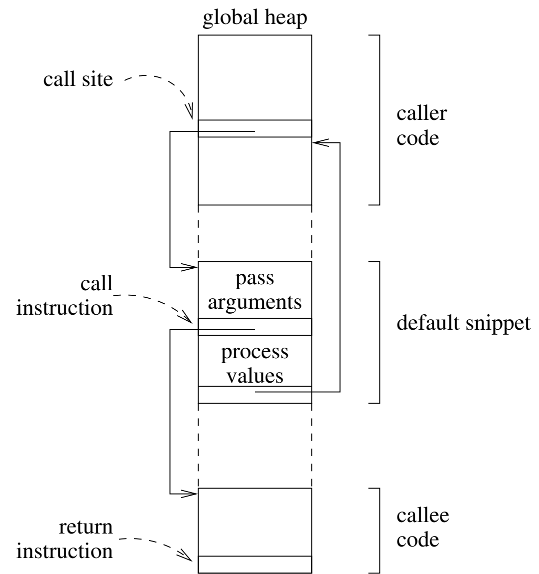

Figure 2: Default snippet.

4.2 Ordinary function form

The code emitted by the caller for a function call consists of a single unconditional *jump* instruction. The target address in that instruction is altered by the callee according to its structure. The code for the function call is contained in an object that we call a *trampoline snippet*, or just *snippet* for short. The callee allocates an appropriate snippet in the global heap as described in Section 3, at some available location, and the unconditional jump instruction of the caller is modified so that it performs a jump to the first instruction of the snippet. The constellation of caller, callee, and snippet is illustrated in Figure 1. We omitted an explicit indication of a control transfer from the snippet to the callee code, because such a control transfer is not always required.

When the callee changes in some way, a new snippet is allocated and the jump instruction is altered to refer to the position of the new snippet. The old snippet is then subject to garbage collection like any other object. For the callee to be able to alter the caller this way, a list of all call sites must be accessible from the name of the callee. For an ordinary Common Lisp implementation, the symbol used to name the callee can store such a list. In SICL, this information would be kept in the data structure describing the callee in the first-class global environment [7]. Either way, to avoid memory leaks, the call site should be referred to through a weak reference.

When code containing a caller is loaded into the global environment, and that caller contains a call site that refers to a function that is not defined at the time the caller is loaded, a *default snippet* is created. The default snippet contains the same instructions that a traditional compiler would create for a call to a function that might be redefined in the future. Thus, the default snippet contains code to put arguments in places dictated by the calling conventions, and it accesses return values from predefined places. It also accesses the function indirectly, either through a symbol object (as most Common Lisp systems probably do) or through a separate *function cell* as described in our paper on first-class global environments [7]. The default snippet is illustrated in Figure 2. The default snippet is also used when the definition of the callee changes, as described below. A default snippet for each call site can either be kept around, or allocated as needed. The former situation is advantageous for a callee with many call sites and for callees that are frequently redefined, as it decreases the time to load a new version of the callee.

In order for the callee to be able to adapt the snippet to its requirements, the caller, when loaded into the executing image, must provide information about its call sites to the system. Each call site contains information such as:

- The name of the callee.
- The number of arguments.
- The type of each argument. If the type is not known, it is indicated as `t`. When an argument is a literal object, its type is indicated as `(eql ...)`.
- For each argument, whether the argument is boxed or unboxed.
- For each argument, its location. The location can be a register or a stack position in the form of an offset from a frame pointer.
- The number of required return values, or an indication that all return values are required, no matter the number.
- In case of a fixed number of return values, for each such value, some limited information of the *type* of

each value. See below for a more elaborate explanation of the restrictions involved for this information.
- Also, in case of a fixed number of return values, for each such value, the location where the caller expects the value.
- Indication as to whether the call is a *tail call*, in which case the snippet should deallocate the frame before returning.

A callee can take advantage of this information to customize the call. The default action is to generate a snippet that implements the full function-call protocol, without taking into account information about the types of the arguments.

While our technique allows for information provided by the caller to be taken into account by the callee in various ways, the opposite direction is not generally possible. The reason is that the callee can change or be redefined in arbitrary ways, and the caller code is fixed, so it can not adapt to such changes in the callee. The only place where some limited amount of adaptation is possible is in the snippet, after the callee code returns.

A direct consequence to this one-directional dependency is that the caller can not, in general, dictate the type of the return values. The current callee will produce the values that its code dictates, no matter what the caller needs. However, it is quite advantageous to be able to return unboxed values of certain types; in particular full-word floating point numbers. For that reason, we allow some restricted type information to be provided by the caller. Thus, if the caller indicates a type other than `t` for some return value, it has to be one of a small number of fixed types, for example `double-float`, `character`, `(signed-byte 64)` and `(unsigned-byte 64)` (assuming a 64-bit architecture). When one of these types is indicated by the caller, the meaning is that the caller requires an unboxed value of this type. Then, if the callee cannot supply such a value, code is generated in the snippet to signal an error.

When a modification is made to a callee that alters its semantics, care must be taken so as to respect the overall semantics of all callers. In particular, a callee can be removed using `fmakunbound` or entirely replaced using `(setf fdefinition)`. In that case, the following steps are taken in order:

(1) First, every call site is de-optimized, which means that a default snippet is allocated for each caller, or the kept default snippet is reused. The unconditional jump instruction is modified to refer to the default snippet. As previously explained, this snippet contains code for the full function-call protocol, and in particular, it accesses the callee using an indirection through the function cell.
(2) Next, the callee is atomically replaced by a new function, or entirely removed by a single modification to the contents of the function cell.
(3) The new function is attached to the list of call sites, and, depending on the nature of the new function, new snippets can then be allocated in order to improve performance of calls to the new function.

When new snippets are substituted, actions may be needed to ensure that that processors do not use stale code. Depending on the types of processors involved, such actions include flushing instruction caches and prefetch pipelines.

The thread responsible for redefining the callee, blocks until step 1 is accomplished. Without this blocking, some callers may get the old version of the callee and some others the new version, thereby violating the overall semantics of a function redefinition. In some cases, it may be acceptable for different callers to get different versions, but in the general case, i.e., when it is observable which versions are used, it is not acceptable. Because of this requirement, redefining a function can be an expensive operation, but redefining a function is expected to be infrequent compared to calling it.

Step 3, on the other hand can be accomplished asynchronously, and even in parallel with caller threads, provided that appropriate synchronization prevents subsequent simultaneous redefinitions of the callee.

4.3 `funcall` with known function name

There are two subcases for this type of form:

(1) The first argument is a special form `quote` with the argument being a symbol. This case can occur as a result of the programmer wanting to avoid capture of the function name, and make sure the name refers to the function with that name in the global environment. It can also occur in the expansion of a macro form.
(2) The first argument is a special form `function`. This case typically occurs as a result of expanding a `setf` macro form and the function name is then of the form `(setf` *symbol*`)`. The expansion uses `funcall` simply because a function name of this form can not be used as the operator of a function form. This case can also occur in the expansion of a macro form.

To handle this case, the compiler treats the symbol `funcall` as a special operator. If the first argument corresponds to any of the two subcases, then the call is treated in the same way as an ordinary function form. Otherwise it generates a call to the function `funcall`.

4.4 `apply` with known function name

As with `funcall`, the same two subcases exist for `apply`, and for the same reason. Again, the compiler treats the symbol `apply` as a special operator and generates a call to the function `apply` whenever the first argument is neither the special form `quote` nor the special form `function`.

However, the case of `apply` is of course more complex than that of `funcall`. Recall that `apply` takes at least two arguments. The first argument is a function designator as with `funcall`. The remaining arguments represent a *spreadable argument list designator*, which means that the last argument is treated as a list of objects, and the arguments to the callee are the objects in that list, preceded by the remaining arguments to `apply`, in the order that they appear.

A very common subcase of this case is a call to `apply` with exactly two arguments. It is used when the execution of some code results in a list of objects, and these objects must be passed as the arguments to some function, in the order that they appear in the list. For this subcase, our technique can be used to avoid the indirection to find the callee entry point as usual. But it can also be used to access the callee arguments directly from the list of objects, so as to avoid unpacking the list to locations dictated by the full call protocol.

A more interesting subcase is that of some intermediate function wanting to override some, but not all of the keyword arguments that it was passed, before calling the callee. The remaining arguments to `apply` are then typically keyword/value pairs. Our technique can then be used to avoid scanning the last argument to `apply` for these keyword arguments. Recall that the standard allows for multiple occurrences of the same keyword argument in an argument list, and that the first occurrence is then the one that is used.

Call-site information resulting from a call to `apply` must be indicated as such, so that the call-site manager can process the arguments as a spreadable argument list designator, rather than as an ordinary suite of arguments.

5 BENEFITS OF OUR TECHNIQUE

Our technique makes possible several features that are not possible when a function call is created by the caller, without knowledge about the callee.

For starters, at least one indirection can be avoided, thereby saving a memory access. When the call is generated by the caller, there must be an indirection through some kind of *function cell*, unless the callee is a function that is known never to change. This indirection is required so that a redefinition of the callee is taken into account by the next call. A typical Common Lisp implementation uses a symbol (the name of the function) for this indirection, whereas SICL uses a separate `cons` cell, but the cost is the same. With our technique, when a callee is altered, the snippet is modified. As a result, no indirection[6] is required. Furthermore, in SICL all functions are standard objects, which requires another indirection from the *header object* to the so-called *rack* where the entry point is stored.

A more significant benefit than saving an indirection is that argument parsing can be greatly simplified. Even in the simple case where all parameters are required, it is no longer necessary for the caller to pass the argument count, nor for the callee to check that it corresponds to the number of parameters. But the advantages are even greater in the presence of optional parameters and in particular for keyword parameters. In a typical call with keyword parameters, the keywords are literals. The argument list can then be parsed once and for all when the snippet is created, and the arguments can be directly copied to the locations required by the callee. This possibility largely eliminates the need for separate compiler macros, as the purpose of a compiler macro is precisely to take advantage of some known structure of the list of argument, in order to substitute a call to a specialized version of the callee.

[6]Though, the snippet is itself a kind of indirection, of course.

The specialized function call can admit unboxed arguments. Avoiding boxing is particularly useful for applications that manipulate floating-point values that are at least the size of the machine word, say IEEE double or quadruple floats in a 64-bit system. When a general-purpose function-call protocol is used, each such argument must be encapsulated in a memory-allocated object before the call, and often, the argument will immediately be unboxed by the callee for further processing.

Return values benefit from the same advantages as arguments. Often, the number of values required by the caller is known statically. The callee can then specialize the transfer of those values to the right locations in the caller. And if the caller requires fewer values than the callee computes, the callee can sometimes be specialized so that extraneous return values do not need to be computed at all. As with arguments, return values can be unboxed, again avoiding costly memory allocations.

When the callee is a generic function, a specialized discriminating function can often be created, provided that enough type information is made available by the caller for the arguments that correspond to specializers of some methods of the generic function. In the extreme (but common) case where the callee is a slot accessor and the class of the specialized argument is known, the snippet can contain the full code to access the slot, without any need to call a particular method function.

Often, *inlining* is used to improve the performance of function calls, either by the application programmer or by the system itself. But inlining some function necessarily increases the code size of each caller of that function. Furthermore, the semantics of inlining are such that the caller must be recompiled for a modified callee to be taken into account. Our technique can often provide enough performance improvement to make inlining unnecessary. Total code size will then be smaller, and the disadvantage of inlining with respect to callee redefinition is eliminated.

Compared to the so-called *ctor* technique describe in Section 2.2, our technique is more general, since it does not involve any source-code transformations. Thus, it can be used with functions defined by the application programmer, and that can change at any point after the callee has been compiled. Furthermore, the *ctor* technique still requires at least one, probably two, indirections (one to access the funcallable object and another one to access the entry point). However, our technique in itself can of course not accomplish the entire optimization machinery required to optimize a function such as `make-instance`, as knowledge of its semantics is required for such optimization.

6 DISADVANTAGES OF OUR TECHNIQUE

The proposed technique is fairly complicated. In order for all the advantages to be had, the callee must be represented in

such a way that multiple versions can be created, depending on different information provided by each caller. On the other hand, most of the benefit of this technique can be obtained with a limited amount of such flexibility. Bypassing argument parsing in the presence of optional or keyword parameters will already provide great benefits. For this benefit to be as useful as possible, it is advantageous to compile a callee in two parts; one part that allows for its parameters to be positioned in any places (registers or stack frame locations) that makes the remaining code as fast as possible, and one part that parses arguments from their default locations into those places. The callee can then generate code for the snippets that moves the arguments passed by the caller to those final places.

The garbage collector must not reclaim snippets that are currently in use, and "in use" can mean that a callee has an activation record on the call stack, so that the snippet can not be reclaimed until the activation record is removed from the stack. As a result, a modification to the garbage collector is required, and code for garbage collectors is notoriously hard to get right.

The technique involves the creation of two unconditional *jump* instructions; one from the core code of the caller to the snippet and another one from the snippet back to the core code of the caller. These additional instructions must be executed, which may use up processor cycles. However, on most modern processors, unconditional jumps are very fast [6].

Finally, there may be some slightly increased probability of contention in the instruction cache, due to the fact that snippets are allocated wherever the global memory manager can fit them.

7 CONCLUSIONS AND FUTURE WORK

We have presented a very general technique for call-site optimization. Our technique subsumes (entirely or partially) several other techniques such as inlining, compiler macros, sealing.

Our technique is very general, and promises several advantages to function-call performance that can not easily be obtained with other techniques. The flip side is that our technique is fairly complicated and requires significant support from both the compiler and the memory manager.

Furthermore, we have not implemented the suggested technique, and the state of the SICL system is not yet such that it can be done soon. The most urgent future work, then, is to create a native SICL executable. We are probably several months away before this work can be accomplished.

8 ACKNOWLEDGMENTS

We would like to thank Frode Fjeld and David Murray for providing valuable feedback on early drafts of this paper. We would also like to thank Duane Rettig for his remarks, and for valuable information about several optimization techniques used in Allegro Common Lisp.

REFERENCES

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

[2] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM, 1984. ISBN 0-89791-125-3. doi: 10.1145/800017.800542.

[3] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, March 1993. URL http://www.acm.org:80/pubs/citations/proceedings/plan/158511/p113-doligez/.

[4] David Gudeman. Representing Type Information in Dynamically Typed Languages. Technical Report TR 93 27, Department of Computer Science, The University of Arizona, October 1993.

[5] Marco Heisig. Sealable Metaobjects for Common Lisp. In *Proceedings of the 13th European Lisp Symposium*, ELS '20, pages 26 – 32, April 2020. URL http://www.european-lisp-symposium.org/static/proceedings/2020.pdf.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055.

[7] Robert Strandh. First-class global environments in common lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.