The Space Overhead of Customization

Sylvia Dieckmann and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/oocsb

Technical Report TRCS 97-21
December 3,1997

Abstract. Customization aims to improve the performance of pure object-oriented languages by compiling multiple copies of a source method, each of them specialized for a certain revceiver type. Like other code duplication techniques, it gains performance by trading code space for better speed. Unfortunately, customization can significantly increase code space, especially for larger programs. We show that customization increases memory usage by almost a factor of three for some applications in the Self-93 system.

We analyze and quantify the factors that lead to this space overhead and identify strategies to eliminate most of it. We focus on dynamically-compiled systems like Self-93 where it is impractical or undesirable to use whole-program analysis or programmer-directed profiling to guide customization decisions. Our experiments show that a combination of relatively straight-forward strategies can bring the code space consumption of customization to within 34% or less of a completely non-customizing system. Thus, even in dynamically-compiled systems customization and efficient memory usage need not be mutually exclusive.

1. Introduction

Pure object-oriented languages such as Self [US87] or Smalltalk [GR83] are difficult to implement because of the high frequency of virtual method calls [DS84], [CUL89]. Much previous work has attempted to overcome this performance problem with a suite of optimization techniques aimed at reducing the cost of dynamic dispatch: inlining [CU93], customization ([CU89], [CU93]), splitting ([CU93], [CU90]), type inference [APS93], type feedback [HU94a], specialization [DCG95a], and adaptive optimization [HU94a]. Together, these optimizations can dramatically improve the performance of pure object-oriented languages.

Many of these optimizations trade code space for speed by creating multiple specialized copies of a piece of source code, each of which can execute faster than code handling the general case. For example, inlining specializes a method to a particular call site, and splitting duplicates part of a method in order to avoid dispatch [CU93]. Similarly, customization compiles multiple versions of a single source method, each of them specialized to a specific receiver type. It especially benefits pure object-oriented languages, since it exploits the fact that there many messages within a method are sent to self, the receiver of the method. With customization, the compiler knows the type of self in the customized copy at compile time and can subsequently inline sends to self, eliminating a common source of dynamic dispatches. Figure 1 shows how a polymorphic send of lessThan: is inlined after the calling method has been customized.

By creating multiple specialized copies, customization increases the size of the compiled code. The resulting code space overhead can be quite substantial. Figure 2 shows that customization increases code size by almost a factor of three for some applications in the Self-93 system. Is this large overhead inherent to customization, or can it be reduced to a more reasonable fraction?

In this paper we identify the factors that contribute to the code size increase and quantify their influence. Several experiments not only identify the sources of unnecessary code duplication but also suggest strategies to eliminate most of these cases. Based on measurements of the Self-93 system, we estimate the success of each strategy in reducing the space overhead.

The results show that a large fraction of the code overhead is caused by customization of unoptimized code. Eliminating this portion can already reduce the memory usage by 17-45%, most likely without any performance loss.

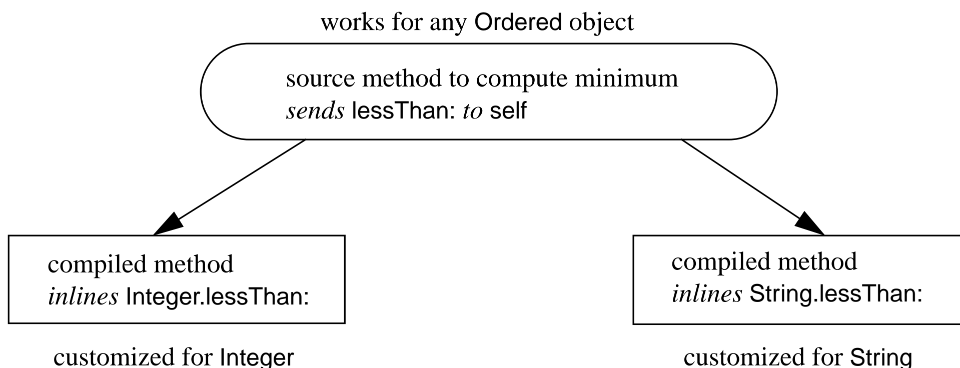

Figure 1. Customization of min:

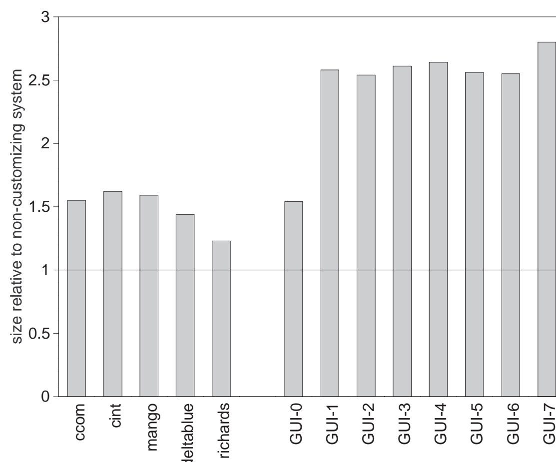

Figure 2. Impact of customization on code space in Self-93

The experimental setup and the benchmarks are described in detail in Section 3

Additional restrictions on the customization of optimized code further reduce space usage to approximately 40% of the size of the current Self system (50% on average for the larger benchmarks). Such a system reduces the space overhead of customization to 34% or less over a completely non-customizing system. Based on these results, we believe that customization and economical memory usage need not be mutually exclusive, and that a dynamically-compiled system can take advantage of the performance benefits of customization without sacrificing compact code.

Dean et al. [DCG95a] addressed a similar problem in Cecil and suggest selective customization based on profile information and global call-graph analysis to suppress unnecessary customization. While their result confirms our basic assumption that most of the benefit of customization can be gained by duplicating only a few methods, their technique to detect these methods is not directly applicable to an interactive, dynamically-compiled system such as Self-93. (See the related work section for a more detailed discussion.) In contrast, our work investigates techniques that reduce the memory impact of customization but do not require global code analysis or profiling, which both could impair the overall performance and responsiveness of a dynamically-compiling system. To our knowledge, this study is the first to address the space overhead of customization in a dynamically-compiled system.

The remainder of this paper is structured as follows. Section 2 contains a review of customization as well as background information about Self and its runtime system. The next four sections study the memory behavior of Self-93. They describe the experimental setup (Section 3), identify sources of excessive customization (Section 4), suggest strategies to overcome these problems (Section 5), and estimate the impact onto runtime performance (Section 6).

2. Background

2.1 Customization

Self is a pure, dynamically typed object-oriented language [US87]. All data are objects and are accessed only by message sends. To overcome the performance problems posed by frequent dynamically-dispatched calls, Self-93 uses customization and other optimizations to eliminate most of the runtime message sends.

Customization allows the compiler to determine the types of many message receivers in a method [CUL89]. It extends dynamic compilation by exploiting the fact that many messages within a method are sent to self, the receiver of the current method. The compiler creates a separate compiled version of a given source method for each receiver type (Figure 1). For example, it might create two separate methods for the min: source method computing the minimum of two Ordered objects (i.e., objects implementing a lessThan: method that imposes a total order on the objects). The polymorphic nature of this method allows it to be reused for many different kinds of objects, e.g., integers, floating-point numbers, strings, etc. If there was only one compiled method for min:, it would have to invoke lessThan: through another dynamically-dispatched call since the receiver could be any Ordered object. Duplicating the compiled code allows the compiler to customize each version to the specific receiver type. Code duplication by itself does not provide any benefits, but knowing the exact type of self at compile time enables the compiler to inline all sends to self. In our example, the two compiled versions of min: can both inline the respective lessThan: method, avoiding the expensive additional call. Customization is especially important in pure object-oriented languages like Self or Smalltalk where so many messages are sent to self, including instance variable accesses and many kinds of user-defined control structures.

2.2 Overcustomization

While customization can significantly improve performance in many situations [CU93][Cha92], it can also lead to excessive duplication of inherited methods. For example, consider a hierarchy of graphical objects whose abstract superclass defines the method canHaveSubmorphs. The default implementation returns true and is inherited by all graphical objects that can contain subobjects. Customization forces the system to compile a separate compiled version of canHaveSubmorphs for each inheriting type of graphical object. Since the code does not benefit from customization (it just returns true), that code is unnecessarily duplicated many times (Figure 3). This situation is called *overcustomization*.

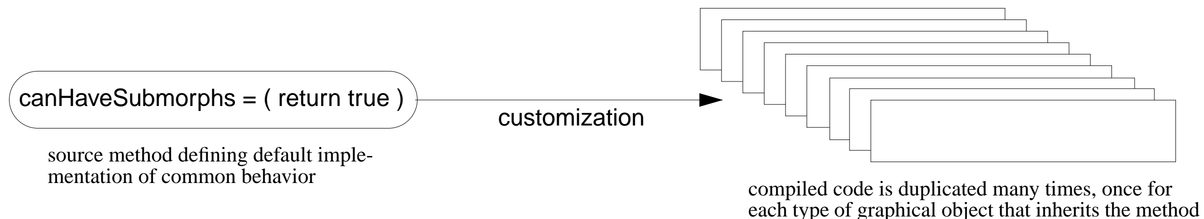

Figure 3. Overcustomization

In large software systems, overcustomization can have a significant impact on code space. In early Self systems which did not have a large code base, this effect was not apparent. But in later systems (Self-93 contains about 1,800 kinds of objects and over 30,000 methods), code size became a problem. For example, in the Self user interface [SMU95] all graphical objects inherit from the morph object defining the functionality common to all morphs. Currently, the common morph interface includes 169 functions which are inherited by 106 concrete morph types, creating a maximum of 169 * 106 = 17,914 customized morph functions to implement the common behavior. In reality, the compiler creates only a fraction of the possible versions since not all methods are used with all receiver types; with dynamic compilation, code is only generated for cases that are actually used. Nevertheless, the problem of overcustomization is real, as demonstrated by the factor of 2.8 code growth observed for the GUI benchmark.

2.3 The Self Runtime System

The Self virtual machine (VM) is based on *dynamic compilation*. Rather than compiling the whole system up front, small units are compiled on demand, interrupting execution. Whenever a certain method/receiver combination is used

for the first time, the VM invokes a compiler and stores the compiled code. In order to avoid long interrupts, the VM uses a “fast but dumb” non-optimizing compiler called NIC (*non-inlining compiler*) for this initial compilation.

The unoptimized code contains invocation counters to keep track of its usage. Whenever a counter exceeds a given threshold, the VM invokes the recompilation system to decide whether to recompile the method with a smarter but more expensive optimizing compiler dubbed SIC (*simple inlining compiler*).

Compiled methods are stored in a code cache. The cache contains both unoptimized (NIC-compiled) and optimized (SIC-compiled) methods. All compiled methods are customized to one specific receiver type. They are retrieved dynamically through a lookup based on the method selector and the receiver type.

3. Experiments

This chapter describes the experiments performed to study space consumption of customization. Our goal was to quantify the memory cost of customization, to identify improvements, and to quantify the space impact of such improvements. In particular, we investigate the space usage of systems that customize only a subset of compiled methods. For every configuration, we simulate the code cache of the modified VM after executing one of the benchmarks.

We only simulate the customization variants, i.e., we did not actually modify the Self VM to produce non-customized code. The main reason for this was that the current Self implementation does not allow non-customized methods. Permitting non-customized methods in the code cache would require substantial changes throughout the VM (e.g., lookup system, deoptimization/debugging system, and recompilation system). Such changes would require a large implementation effort, and they would also affect the overall performance of the system. Also, the non-customizing system would likely be less tuned and therefore less efficient, which could distort measurements of the performance impact of customization. By simulating the partially customized systems, we can study a variety of configurations with only moderate implementation effort. Although the simulations ignore execution performance, they accurately capture the code space behavior of a real system, and thus we hope they will be helpful to implementors contemplating the incorporation of customization into a dynamically-compiling system.

3.1 Experimental Setup

In order to measure the memory performance of customization, we executed the benchmarks application using an instrumented version of the Self-93 system. During the benchmark runs, the VM periodically takes a snapshot of its code cache and writes the data to disk. This data captures a static picture of the compiled code generated for the particular benchmark. We then used off-line analysis tools to determine, for example, how many of the compiled methods are unoptimized, or how often a particular source method is customized. Being a static picture of the code cache, the logged data does not reveal, for example, exactly when or why a method was compiled. However, for some benchmarks we also compared consecutive snapshots in order to observe, for example, how recompilation of NIC methods changes the code cache over time.

The study is based on experiments with two groups of benchmarks (see Table 1). Figure 4 and Figure 5 present a summary of the code space after executing each benchmark 200 times in Self-93. We will use these numbers as a baseline for the interpretation of subsequent experiments.

benchmark	description	source code (lines)[a]	compiled code (Kbytes)[b]	degree of customization
ccom	Cecil-to-C compiler compiling the Fibonacci function (the compiler shares about 80% of its code with the interpreter, cint)	11,500	1262	medium
cint	interpreter for the Cecil language running a short Cecil test program	9,000	475	medium
mango	automatically generated lexer/parser for ANSI C, parsing a 700 line C file	7,000	459	low
deltablue	two-way constraint solver developed at the University of Washington	500	98	low
richards	simple operating system simulator originally written in BCPL by Martin Richards	400	52	low
GUI	Self user interface [SMU95]	25,000	3,600	high

Table 1. Standard benchmark programs

[a] excluding blank lines
[b] size of the code cache when executing with Self-93

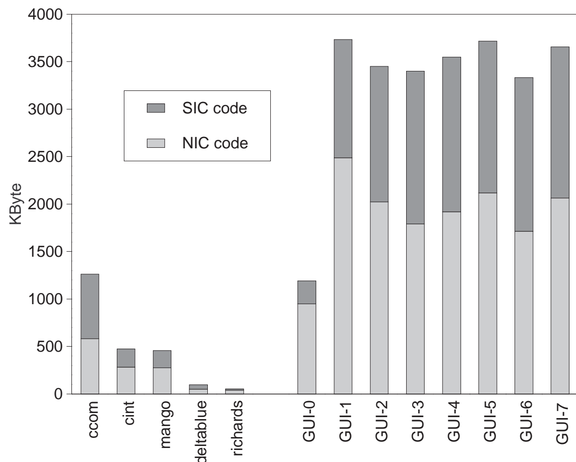

Figure 4. Total size of compiled methods in Self-93

3.2 Standard Benchmarks

The first benchmark group consists of five Self programs, each of which shows a distinct and highly repeatable behavior. Figure 4 shows that the benchmarks differ quite a bit in code size. On the lower end, richards spends its time in very few methods; even after several repetitions, its code cache does not grow over 52 Kbytes, with only 22% being optimized code. A profile shows that richards spends almost no time in unoptimized code once the runtime system has recompiled all hot methods. In the optimized portion of richards' code cache, no method is customized to more than two receiver types. ccom on the other end is more complex and significantly larger. At the end of its execution, the code cache contains 1,266 Kbytes, of which 54% is optimized code.

In general, a single run of a benchmarks did not execute long enough to fill the cache and optimize the important calls. We therefore invoked each benchmark 200 times before measuring the code cache, thus making sure that the system has enough time to reach equilibrium. At the end of the 200th iteration, we created a snapshot from its cache contents.

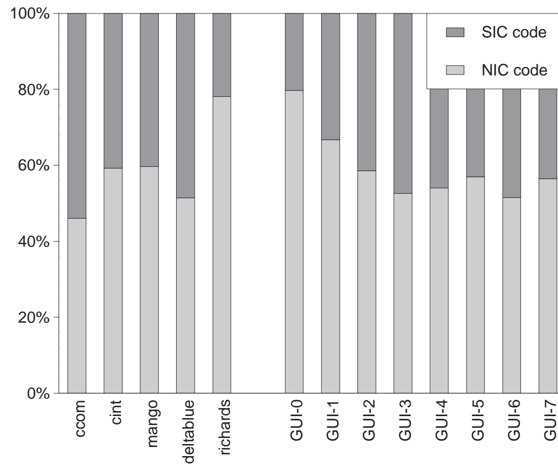

Figure 5. Ratio of optimized vs. unoptimized code in Self-93

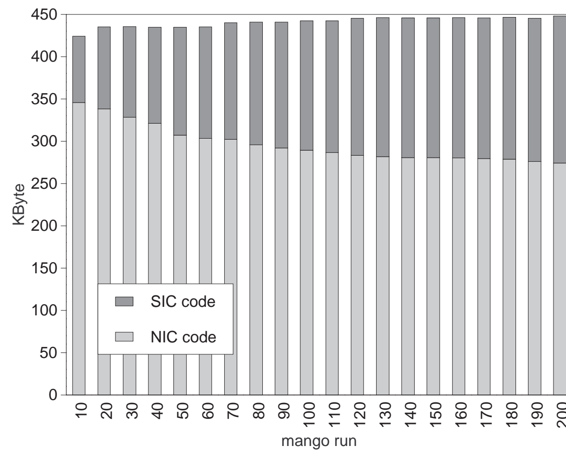

Figure 6. Code space evolution during 200 executions of mango

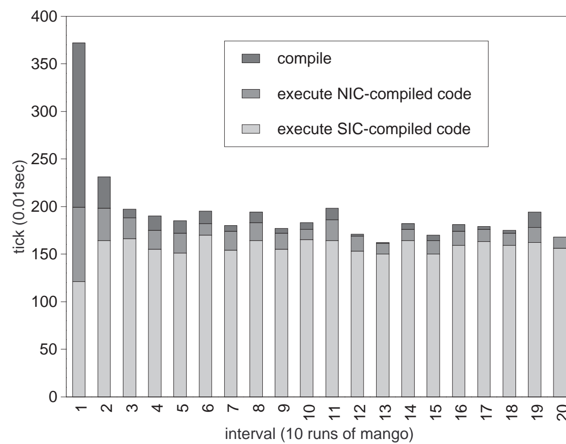

Figure 7. Time spent for execution and compilation during 200 runs of mango

Although most of the data presented in this paper is based these snapshots of the final code space state, we also measured how the code space configuration evolves during the 200 consecutive runs. For these experiments, the VM profiled every subgroup of 10 iterations[1] and recorded a snapshot of the code cache after every 10th run. Figure 6 and

Figure 7 show the results for mango. In the beginning, the VM spends most of its time executing unoptimized code and compiling. The code cache is full with unoptimized code. After a while, the recompilation system starts replacing unoptimized methods with optimized code. Although the overall size still increases, the cache now contains much less unoptimized code. Soon, the compiler is invoked only infrequently. After 30-40 repetitions, the system has reached equilibrium; a longer series of repetitions would not improve the quality of the code.

3.3 User Interface Benchmark

To complement the standard benchmarks, we also measured a much larger and highly polymorphic program, the Self graphical user interface (GUI) [SMU95]. It consists of over 25,000 lines of highly factored Self code, including a large hierarchy of user interface elements called morphs. Morphs are used at all levels of the user interface, from basic elements such as characters and lines to more complicated ones such as menus and buttons. All in all, the GUI contains more than 100 different kinds of morphs. Since they all inherit at least some default behavior, the program is ripe for customization.

For the measurements, a user was working interactively with the Self GUI for over an hour. The actions involved searching through the system, changing graphical objects, starting an animation, etc. During the session, we took several snapshots of the code cache, one right after start-up (GUI-0)[1] and the others roughly every 10 minutes (GUI-1 to GUI-7).

Based on our knowledge of the program's structure, we expected the code cache to contain highly customized code at the end of this session. Our measurements confirm this. Figure 4 and Figure 5 show large code caches with a high fraction of optimized code. All except the very first snapshot contain methods customized for over 50 receiver types. Even just starting the GUI requires almost as much compilation as a whole series of ccom runs.

4. Identifying the Overhead of Customization

4.1 The Overhead of Customization

Our first task is to quantify and analyze the costs of customization in Self-93. We start by determining the code size of a completely non-customizing system, where no method is ever customized. This hypothetical system serves as a reasonable lower bound on code size. It is not a strict lower bound since it is theoretically possible to reduce the code size with customization. For example, a customized method may be more optimized and thus smaller than the non-customized version, resulting in a net code size decrease if the method is customized only once.

benchmark	non-customized (Kbytes)	Self-93 (Kbytes)	factor
ccom	815	1262	1.5
cint	294	475	1.6
deltablue	68	98	1.4
mango	289	459	1.6

Table 2. Code size of a non-customizing system compared to Self-93

[1] All timings exclude any overhead introduced by the measurement process (e.g., the snapshotting of the code cache contents).

[1] Since the first data point (GUI-0) shows a atypical setting found only during start-up, this point is excluded from the computation of average values.

benchmark	non-customized (Kbytes)	Self-93 (Kbytes)	factor
richards	43	52	1.2
GUI-0	775	1192	1.5
GUI-1	1446	3734	2.6
GUI-2	1358	3452	2.5
GUI-3	1305	3399	2.6
GUI-4	1343	3549	2.6
GUI-5	1451	3717	2.6
GUI-6	1304	3332	2.6
GUI-7	1305	3655	2.8

Table 2. Code size of a non-customizing system compared to Self-93

Table 2 shows the current code sizes relative to the estimated sizes produced by a non-customizing system. In such a system, not more than one optimized and one unoptimized method for the same source method are allowed to exist simultaneously in the code cache.[1]

For the standard benchmarks, the presence of customization increases code size by a factor of 1.2-1.6. For these programs, the extent of customization is modest; many methods have only a single receiver, and none of the benchmarks contains any optimized method customized to more than eight receivers.

The larger GUI benchmark, on the other hand, suffers significantly from code expansion due to customization, requiring almost three times more space when executed with a customizing system. The GUI benchmark underscores a disturbing trend: the larger the application, the higher the customization overhead. In particular, the two smallest programs (deltablue and richards) show the smallest overhead, and GUI (the largest program by far) exhibits the largest overhead by far. For customization to scale to larger systems, it is clearly desirable to throttle it in some way in order to avoid this code space explosion.

4.2 Customization Overhead of Unoptimized Code

We now analyze how much of the code added by customization is still unoptimized. Figure 8 and Figure 9 show the current system's code sizes, broken down into optimized and unoptimized code. The lower two segments of each bar

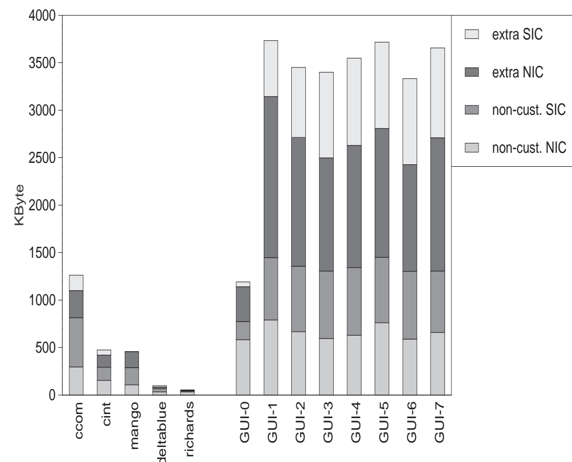

Figure 8. Amount of optimized and unoptimized code in Self-93

(min-cust. NIC and non-cust. SIC) together show the expected cache size when no method is customized; this is the

[1] Section 4.2 describes in detail how we estimate the size of the non-customized code cache.

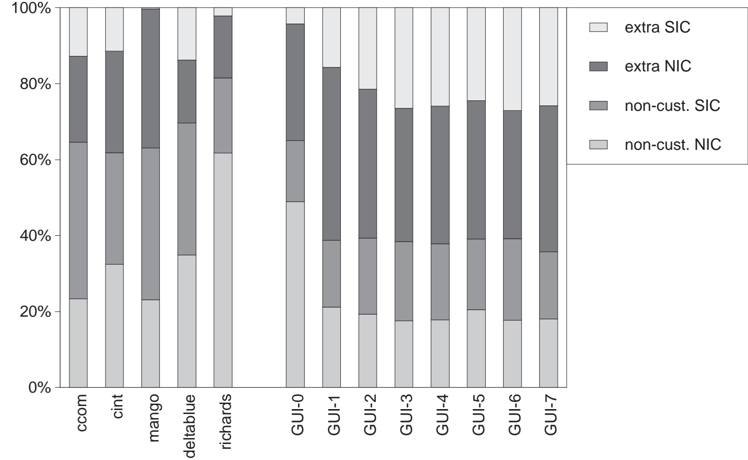

Figure 9. Relative distribution of optimized and unoptimized code in Self-93

baseline used in the previous section. The upper two segments represent the amount of customized code, again split into unoptimized and optimized code. Figure 9 shows the same data expressed as percentages of the overall code size.

Both figures show that a large portion of the overhead introduced by customization is caused by unoptimized code, i.e., code created by the NIC compiler. Extra copies of unoptimized methods consume more space than optimized copies. Depending on the benchmark, unoptimized code accounts for between 55% and 99% of the overhead (75% on average for the standard benchmarks and 61% for GUI). Clearly, reducing the customization of unoptimized method is a promising avenue for code size reduction.

4.3 Estimating the Size of Non-Customized Code

In the graphs above, the absolute code size and the total amount of optimized and unoptimized code correspond to actual numbers measured in Self-93. The baseline numbers for the non-customizing system, on the other hand, are estimated. A system that does not create a special copy for each and every occurring receiver type must compile a general-purpose copy to handle the cases where no customized method is available. The size of this general-purpose method will most likely differ from the sizes of all the customized copies in Self-93. For example, different amounts of inlining might influence not only the size of the code created for the non-customized method itself, but also the code cache as a whole. Since Self-93 requires full customization, we could not actually measure the size of general-purpose code. Instead, we used a heuristic to estimate the sizes.

Our approximations of the size of non-customized methods are based on the actual sizes of compiled methods observed with Self-93. We first examine the code cache, group the compiled methods into optimized and unoptimized code, and determine the peers for every compiled method. Two methods are peers if they are of the same type (optimized/unoptimized) and implement the same source method. For every group we pick one peer to represent the size of the hypothetical non-customized method.

When choosing the representative peer we experimented with three different heuristics, assuming that a non-customized compiled method would have the size of (a) the smallest peer, (b) the largest peer, or (c) the average of all peers in the group. Figure 10 and Figure 11 show the three possible distributions that result from the heuristics show for each benchmark. The differences between the estimates are rather small. For example, the distribution of ccom is 22.9% / 23.1% / 39.6% / 14.4%[1] based on the smallest peer and 23.3% / 22.7% / 43.6% / 10.4% based on the largest peer. All other benchmarks show even smaller deviations. Since the differences are fairly small, we believe that using the size of the average customized copy is an appropriate approximation of the size of the corresponding non-customized method. For the remainder of this paper, we estimate the size of a non-customized copy based on the average size of the corresponding customized instances.

[1] non-customized NIC / extra NIC / non-customized SIC / extra SIC

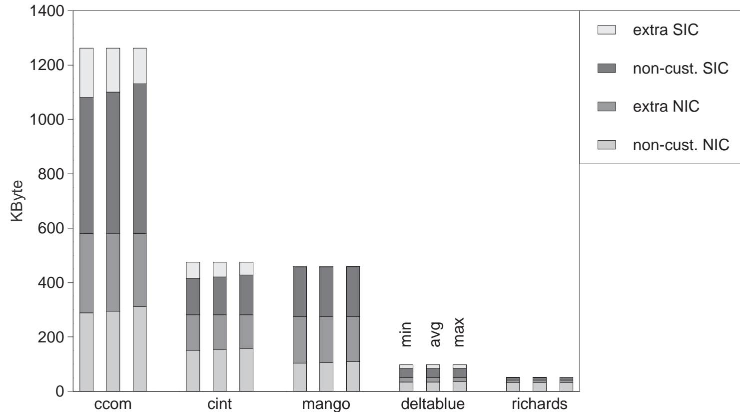

Figure 10. Comparison of code size estimates (standard benchmarks)

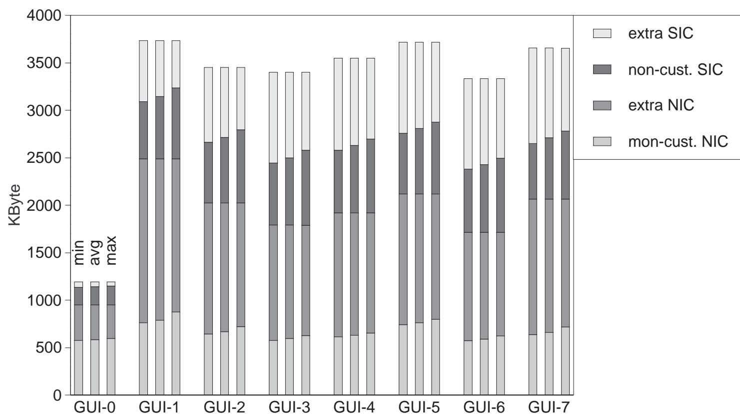

Figure 11. Comparison of code size estimates (GUI benchmark)

5. Reducing the Space Overhead of Customization

This section contains a sequence of experiments that show step by step how to reduce the space overhead introduced by customization. Ultimately, our goal is to identify the ineffective usages of customization whose elimination would reduce the space overhead at the cost of only a minor performance loss. We simulate scenarios with different strategies for controlling customization and determine the expected savings.

5.1 Avoiding NIC Customization

The high percentage of unoptimized (NIC) code that contributes to the overhead of customization is quite surprising, since customization of unoptimized methods does not provide any performance advantage. As discussed before, customization benefits performance only in combination with other compiler optimizations, most notably inlining. Per definition, the NIC compiler does not inline at all, and thus a customized copy compiled with the NIC compiler is identical with its original. In other words, customization of unoptimized methods consumes space but does not improve code quality. (Self-93 customizes unoptimized code because it was much simpler to integrate the NIC into the system this way; although there were plans for allowing non-customized code, they were never implemented because of the need to rewrite substantial amounts of legacy code in the VM.)

Figure 12 shows the code saving achieved by not customizing unoptimized methods (Table A-1 in the appendix provides the raw data). The simple strategy is very successful: for most programs, it reduces code space requirements by a factor of about 1.5. For the standard benchmarks, the remaining customized code is very small, using less than 20% of the overall code space. The code for the graphical user interface, however, still is 60% larger than in a completely non-customizing system.

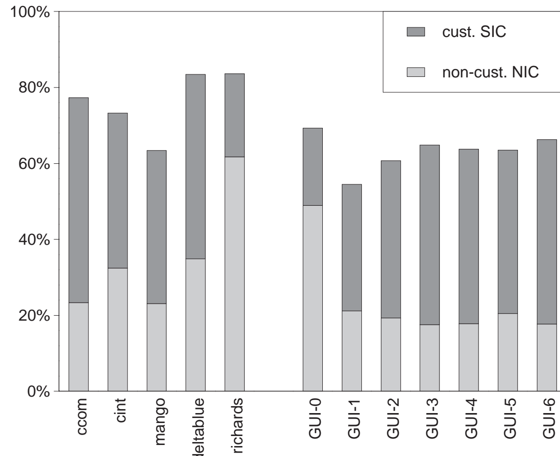

Figure 12. Code sizes with a system customizing only optimized (SIC) code (100% = Self-93)

5.2 Reducing SIC Customization

We can further reduce the space overhead of customization by restricting the creation of customized SIC code. In particular, we can try to eliminate excessive customization by limiting the maximum number of copies customization can create. For example, it is unlikely that creating 50 copies of a method really improves performance, because the cost of compiling these copies, as well as the extra instruction cache misses caused by the 50 times larger code, are likely to negate the benefits of customization. In a system using adaptive compilation, limiting customization is relatively straightforward: with a limit of N copies, the compiler simply customizes for the first N receiver types that exceed the recompilation threshold and then creates a non-customized method to handle all subsequent cases. (Recall that a system using adaptive compilation optimizes methods only if their use exceeds a certain threshold [HU94a].) Most likely, such a strategy will customize code for the most important receivers since these will exceed the recompilation thresholds first.

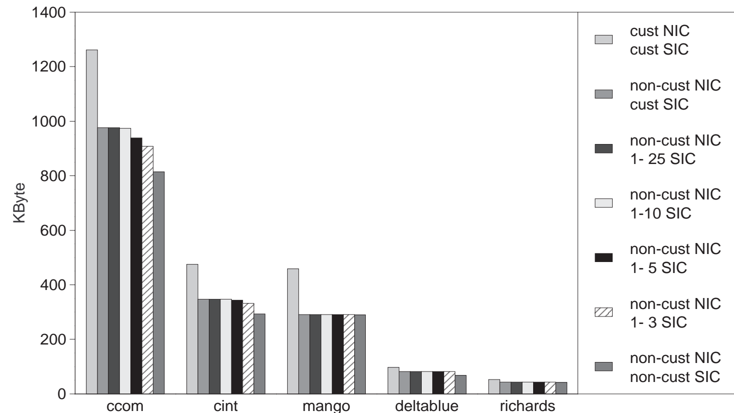

Figure 13. Code sizes when restricting SIC customization (standard benchmarks)

Figure 13 and Figure 14 show the effect of different levels of restricting SIC customization. The two rightmost bars in each group repeat numbers presented in previous sections (fully customized / no customization of NIC code) and are included for better comparison. The other bars show systems that customize only optimized code, and where only a certain number of optimized copies is allowed for each Self source method.[1]

[1] “25 SIC” reads “up to 25 optimized copies per source method”.

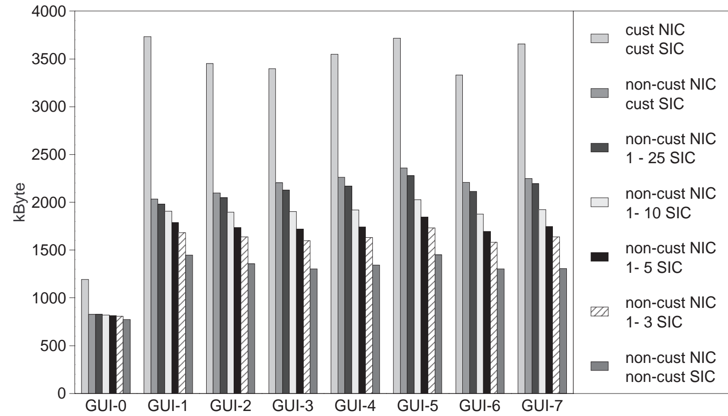

Figure 14. Code sizes when restricting SIC customization (GUI benchmark)

For most of the standard benchmarks except ccom, the influence of restricting SIC customization is very small because there is very little customized SIC code to begin with (as discussed in the previous section). For the GUI benchmark, restricting SIC customization shows a clear improvement. Restricting customization to ten copies per method reduces code size by a factor of up to 1.2 over the system allowing unrestricted SIC customization, and restricting customization to three copies per method results in a factor of up to 1.4 improvement.

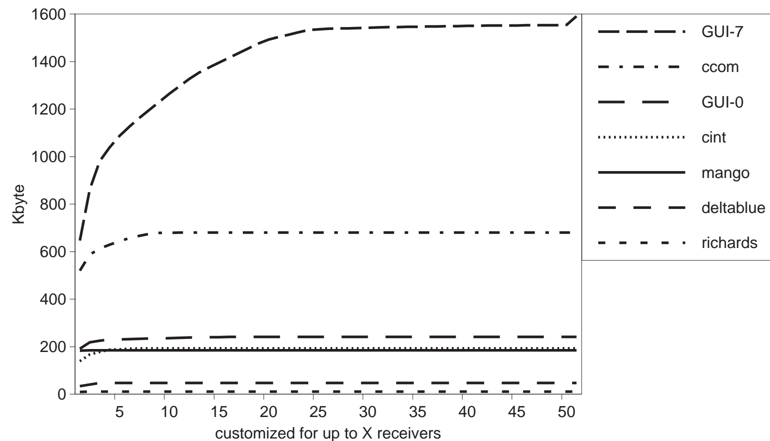

Figure 15. Correlation between size of SIC portion and maximal number of customized SIC copies.

Order in legend corresponds to final size of code portion. The curves for GUI-1 to GUI-6 resemble GUI-7 and are omitted for better readability.

Figure 15 shows a more detailed picture of the amount of customization that takes place in each benchmark. The graph shows code size as a function of the degree of customization. For example, the data points at x=10 show the SIC code sizes of the benchmarks when including only the first 10 optimized copies for every source method. The curves for the standard benchmarks flatten out very early, indicating that they do not contain heavily customized methods. The GUI, on the other hand, is much more customized, flattening out only around 25 copies. But even here the very heavily customized methods do not contribute that much to the code size; the majority of the code space is filled by methods that are customized less than 10-15 times.

5.3 Eliminating Unoptimized Code Entirely

Unoptimized code represents 46-80% of the total code (see Figure 5); even after eliminating NIC customization, the remaining NIC methods still consumes 27-74% of the code (an average of 45% for the standard benchmarks and 31% for the GUI benchmark). In addition, the benchmarks spend very little time executing unoptimized code, as Figure 16 demonstrates.

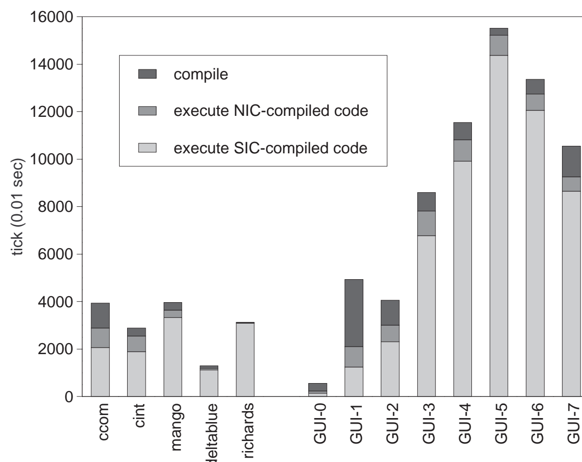

Figure 16. Total time spent for execution and compilation with Self-93

Replacing the NIC compiler by an interpreter would eliminate the unoptimized code in the code cache entirely (at some cost in execution speed, of course). When combined with restricting SIC customization to five copies per method, such a system would reduce the code size by 50-80% compared to Self-93 (Figure 17). However, in this case the code size decrease may not accurately reflect the actual savings since Self's bytecodes would have to be redesigned to allow for efficient interpretation, and this change would probably increase their size.

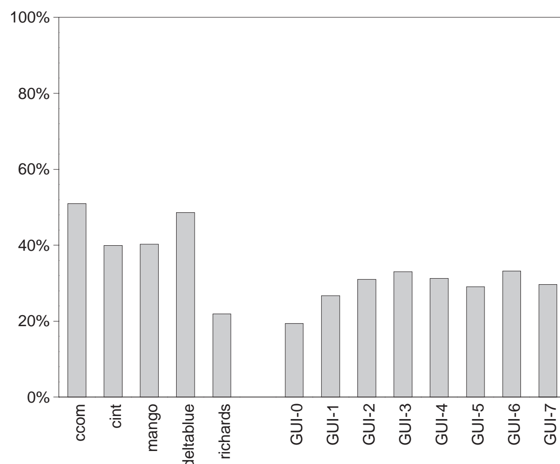

Figure 17. Code sizes with a interpreter and five or less customized copies per method (100% = Self-93)

6. Effects on Runtime Performance

Most of the steps discussed in the previous sections will have some impact on runtime performance since reducing the degree of customization may also reduce the speedup gained by this optimization technique. Unfortunately, the exact extent of this performance impact is hard to determine based on the data we collected. The main goal of our experiments was to determine the impact on memory usage, not performance. However, we can at least estimate the expected performance impact.

The first strategy, not customizing unoptimized methods, is very likely to be performance-neutral or to even improve performance. Customization does not improve code quality at all for unoptimized code, so that a system will not slow down if customization is disabled. To the contrary, it may run faster since the much smaller overall amount of code will

decrease instruction cache misses (Self-93 spent up to 40% of its execution time in instruction cache misses on a SPARCstation-1 [HU94a]).

Since the assumption that all code is customized permeates the runtime system of the Self-93 virtual machine (and all Self VMs before it), many system components would be affected by the introduction of non-customized code. However, except for the lookup system, most of these changes would probably be performance-neutral. Sends from non-customized methods may become more polymorphic since a single compiled method now serves many receiver types, and thus sends from within that method may frequently change their receiver types, reducing the effectiveness of inline caching.

The performance impact of partial customization of optimized code is less predictable since we don't know the amount of time spent in eliminated customized copies. However, we are encouraged by previous studies performed in the context of the Vortex compiler that show that most of the benefit of customization can be achieved with a relatively small amount of customization [DCG95a]. Selective specialization increased the performance of Cecil programs while significantly reducing code space at the same time. Similarly, a study of profile-based customization for C++ found that relatively few methods needed to be customized in order to eliminate most calls that can be eliminated with customization [The96]. Although there are many differences between these systems, it appears likely that a partially customizing system could achieve its space reductions without an undue loss of performance.

7. Related Work

Customization has been applied to several object-oriented languages, including Self ([CU89], [HU94a]), Sather ([LS91]), Trellis ([Kil88]), and C++ ([Lea90], [The96]). Most of these systems combine customization with other optimization techniques (e.g., inlining and method splitting in Self.) Because of the nature of customization—it just enables the compiler to perform other optimizations—the benefit of this technique is sometimes difficult to isolate. Chambers reports a speedup of factor 1.4 - 9 in Self [Cha92], and Lea suggests an order of magnitude speedup for C++ for small programs [Lea90]. Most of the early work on customization focuses on the performance gained with this technique rather than on the code size increase.

Several previous systems include forms of adaptive customization. Cooper et al [CHK92] and Hall [Hall91] specialize Fortran procedures to obtain better information for dataflow optimizations, for example, to parallelize a loop. They present a general framework for identifying beneficial specializations without using profile information. Plevyak and Chien [PC95] implement cloning techniques in the Illinois Concert compiler. They also create clones based on global data flow analysis, but focus on combining clones not required for optimization, thus minimizing the number of clones created.

Dean et al. [DCG95a] present a similar algorithm for Cecil, a pure object-oriented language with multiple dispatch. Cecil uses customization—called specialization—in a more general sense than Self. A method can not only be customized to its receiver but also to its argument types. Therefore, the problem of excessive customization is even more apparent in Cecil and needs to be addressed. The algorithm suggested by Dean et al. is based on a global (program-level) weighted call graph obtained from an execution profile and aims to minimize dynamically-dispatched calls by copying parts of the call graph, starting with the most profitable parts (based on the relative execution frequencies). The algorithm's benefit estimation is defined in terms of the number of eliminated dynamic dispatches.

Although the study's results support our assumption that most of the benefits of customization can be gained by customizing only a few methods, the technique proposed to select the candidates for customization is not directly applicable to an interactive environment using dynamic compilation. The benefit estimation described in [DCG95a] is based on detailed profiling and non-incremental program-wide analysis, both of which are difficult to combine with dynamic compilation as used in Self-93. A dynamically compiling system would have to collect and exploit the data while the application is running, which could have a significant impact on overall performance. For example, Grove et al. [G+95] report overall execution time overheads of up to 50% for profiling. Furthermore, the profile would most likely be very large (several megabytes for Self) and difficult to compress with standard techniques, and the full call graph is not available. Finally, if the required global analysis is non-incremental it could introduce large pauses, thus reducing the responsiveness of the system.

8. Conclusions

We studied the space overhead of customization in the Self-93 system. Although this overhead can be quite large (almost a factor of three in the case of Self's graphical user interface), it can be reduced considerably with relatively simple techniques. For example, suppressing customization of unoptimized code, which most likely has no effect on performance, can improve the memory consumption of our benchmarks by a factor of 1.5 on average.

Limiting the customization of optimized methods can further improve the space cost of customization. For example a system that does not create more than five customized copies for any method can reduce the code space by a another factor of 1.3 for large benchmarks. Together, these techniques reduce the space overhead of customization to 34% or less over a completely non-customizing system. Thus, even a dynamically-compiled systems can combine customization and efficient memory usage.

All of the proposed strategies require no code analysis and no global information or expensive profiling. Instead, they take advantage of the adaptive recompilation system already present to decide at runtime whether and how to optimize a method based on an invocation counter. Although the presence of global information would undoubtedly allow superior solutions, the highly interactive nature of exploratory programming environments such as Self favor solutions that do not require maintaining large data structures and that allow for rapid change.

9. References

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP '93 Conference Proceedings,* p. 247-267, Kaiserslautern Germany, July 1993.

[Cha92] Craig Chambers, *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. Thesis, Stanford University, April 1992.

[CDG95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *17th International Conference on Software Engineering*, Seattle, WA, April 1995

[CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-Program Optimization of Object-Oriented Languages. University of Washington, Technical Report 96-06-02, June 96.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for Self, a Dynamically-Typed Object-Oriented Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* Portland, OR, June 1989. Published as SIGPLAN Notices 24(7), July 1989.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 150-164, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June 1990.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings,* October 1991. Published as *SIGPLAN Notices 26(10),* October 1991.

[CU93] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. *User Interface Software and Technology Conference Proceedings*, Atlanta, GA, November 1993.

[CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in SELF. *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

[CHK92] K.D. Cooper, M.W. Hall, and Ken Kennedy. Procedure Cloning. In *IEEE Intl. Conference on Computer Languages*, p. 96-105, Oakland, CA, April 1992.

[DH88] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience* 18(8): 775-90, August 1988.

[DCG95a] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *PLDI '95 Proceedings*, San Diego, CA, June 1995.

[DCG95b] Jeffrey Dean, Craig Chambers, and David Grove. Optimization of Object-Oriented Programs using Class Hierarchy Analysis. *ECOOP '95 Proceedings*, Aarhus, Denmark, August 1995.

[DS84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[GR83] Adele Goldberg and David Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA, 1983.

[G+95] David Grove, Jeffrey Dean, Charles D. Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. *OOPSLA '95 Conference Proceedings*, Austin, Texas, October 1995.

[Hall91] Mary Wolcott Hall. *Managing Interprocedural Optimization.* Technical Report COMP TR91-157 (Ph.D. Thesis), Computer Science Department, Rice University, April 1991.

[Höl94] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* Ph.D. thesis. Department of Computer Science, Stanford University, August 1994.

[HU94a] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, p. 326-336. Published as *SIGPLAN Notices 29(6)*, June 1994.

[HU94b] Urs Hölzle and David Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. *OOPSLA '94 Conference Proceedings*, Portland, OR, October 1994.

[Kil88] Michael F. Killian. Why Trellis/Owl Runs Fast. Unpublished manuscript, March 88

[Lea90] Douglas Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, p. 301-314, San Francisco, CA, April, 1990.

[LS91] Chu-Cheow Lim and Andreas Stolcke. Sather Language Design and Performance Evaluation. Technical Report TR 91-034, International Computer Science Institute, May 1991

[PC95] John Plevyak and Andrew A. Chien. Type Directed Cloning for Object-Oriented Programs. In *Workshop for Languages and Compilers for Parallel Computers*, Columbus, Ohio, August 1995

[SMU95] Randall B. Smith, John Maloney, and David Ungar. The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity and Flexibility. In *OOPSLA '95 Conference Proceedings*, Austin, Texas, October 1995

[The96] Mark Theiding. Customization in C++. M.S. Thesis, Department of Computer Science, University of California, Santa Barbara, 1996.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, p. 227-241, Orlando, FL, October 1987. Published as *SIGPLAN Notices 22(12)*, December 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

Appendix A Raw data

benchmark	non-customized (Kbytes)	non-cust. NIC + cust. SIC (Kbytes)	Self-93 (Kbytes)	Self-93 / non-customized	Self-93 / non-cust NIC+cust SIC
ccom	814.9	975.7	1261.7	1.5	1.3
cint	293.5	347.7	474.9	1.6	1.4
deltablue	68.0	81.4	97.6	1.4	1.2
mango	289.4	291.0	459.1	1.6	1.6
richards	42.5	43.7	52.2	1.2	1.2
GUI-0	775.1	825.7	1191.7	1.5	1.4
GUI-1	1446.4	2034.3	3733.8	2.6	1.8
GUI-2	1358.0	2096.5	3451.9	2.5	1.6
GUI-3	1304.6	2205.1	3399.2	2.6	1.5
GUI-4	1343.4	2261.5	3549.1	2.6	1.6
GUI-5	1451.4	2360.3	3716.9	.26	1.6
GUI-6	1304.1	2207.4	3331.7	2.6	1.5
GUI-7	1305.2	2249.5	3655.0	2.8	1.6

Table A-1. Code sizes