Chapter 14 Performance Evaluation

We illustrated the new techniques described in this dissertation on small examples in previous chapters, and in that context they appeared to be quite effective. In this chapter we explore the effectiveness of our implementation of these techniques on actual SELF programs by measuring the execution speed, compilation time, and compiled code space consumption of a suite of SELF programs ranging in size from a few lines to a thousand lines. We will evaluate our work from three perspectives:

- How effectively does the combination of these new techniques narrow the gap in performance between pure object-oriented languages such as SELF and traditional languages such as optimized C or optimized Lisp?
- Which of the new techniques are most effective? Which are most costly?
- What are promising areas for future work?

Our answers to these questions are presented in sections 14.2, 14.3, and 14.4, respectively. The next section prefaces these results with a description of our measurement methodology.

14.1 Methodology

14.1.1 The Benchmarks

We analyzed the performance of the SELF implementation using a selection of benchmark programs. These benchmarks include several "micro-benchmarks" gathered from various sources, the Stanford integer benchmark suite [Hen88], the Richards benchmark [Deu88], and several SELF programs originally written without benchmarking in mind. Source code for all these benchmarks is available from the author upon request.

The eleven micro-benchmarks are very short and typically stress only a few aspects of the implementation, such as the speed of integer calculations or the speed of procedure calls. The Stanford integer benchmark suite is composed of the **`perm`**, **`towers`**, **`queens`**, **`intmm`**, **`quick`**, **`bubble`**, **`tree`**, and **`puzzle`** benchmarks, originally collected by John Hennessy and Peter Nye to help design and measure RISC microprocessors and compilers. The Stanford integer benchmarks are larger than the micro-benchmarks, and all exercise integer calculations, generic arithmetic, and user-defined control structures (particularly **`for`**-style loops); all but **`tree`** also stress array accessing. For SELF and Smalltalk, all the Stanford integer benchmarks other than **`puzzle`** were written in two versions: one as similar to the C version as possible, and one taking advantage of the message passing features of SELF and Smalltalk by associating the core of the benchmark code with the data structures manipulated by the code. Both versions of each benchmark perform the same algorithm, with no source-level optimizations made in either version, but the object-oriented version sends more messages to **`self`** and fewer messages to other objects than the procedure-oriented version. Results for the versions of the benchmarks in the more object-oriented style are identified by the **`oo-`** prefix in the benchmark names. Section B.1 of Appendix B describes the individual micro-benchmarks and Stanford integer benchmarks in more detail.

None of the micro-benchmarks or Stanford integer benchmarks is particularly object-oriented. This is to be expected, since we translated them into SELF from traditional non-object-oriented languages such as C. This lack of object-orientation implies that any conclusions that may be drawn from measurements of these benchmarks must be limited to the effectiveness of the SELF implementation at running smaller, more traditional programs. Any conclusions about the performance of SELF on more object-oriented programs based on the performance of these benchmarks must be more tentative and speculative. However, most object-oriented programs contain more traditional portions that must scan through an array or perform simple arithmetic calculations; these small non-object-oriented sections in fact may comprise much of the running time of an application. Therefore, an improvement in performance for non-object-oriented code is likely to improve overall system performance.

The **`richards`** benchmark is a 400-line operating system simulation benchmark originally written by Martin Richards to test BCPL compilers. This benchmark maintains a queue of tasks and spends its time primarily removing the first task from the queue, processing it, and usually appending it to the end of a different queue for further processing; the benchmark begins by initializing the queue with a few tasks and ends when the main task queue is empty. The **`richards`** benchmark is different from the previous ones in that it is large enough not to overly stress any particular operations and has no tight loops or recursions. Instead the **`richards`** benchmark manipulates data structures and therefore is perhaps more representative of typical object-oriented programs than the previous benchmarks. The **richards** benchmark is also unusual in including a message send that actually invokes different methods for different calls: the **runTask** message which is sent to each task after it is removed from the head of the queue and is defined differently for different kinds of tasks.

To measure the effectiveness of the new techniques on truly object-oriented programs, we also measured several SELF programs originally written to be useful in their own right and not as benchmarks. The **pathCache** and **primMaker** programs are in active use today.

- **pathCache** computes a mapping from objects to names inferred from the object structure, used as part of the SELF user interface. **pathCache** is 140 lines long, excluding the code for other supporting data structures such as dictionaries (SELF's key-value mapping data structure).
- **primMaker** generates SELF and C++ wrapper functions for user-defined primitives from a textual description of the primitives. The benchmark was run on a test file that exercises all the different possible descriptions and so ends up being dominated by compile time. **primMaker** is 1000 lines long.
- **parser** parses an earlier version of SELF. The benchmark was run on four relatively short expressions. **parser** is 400 lines long.

These benchmarks use features of SELF such as prototype-based design and dynamic inheritance that are not available in any of the other languages, and so versions of these benchmarks exist only for SELF. These benchmarks cannot be used to compare the performance of SELF to other languages and systems, but they can be used to measure the effectiveness of the various techniques developed for SELF by comparing one configuration of the SELF system against another.

The following table summarizes the benchmarks we measured:

small	eleven micro-benchmarks (1 to 10 lines long)
stanford	seven Stanford integer benchmarks, written in a traditional procedure-oriented style (20 to 60 lines long)
oo-stanford	seven Stanford integer benchmarks, written in a more object-oriented style (different from **stanford** only for SELF and Smalltalk)
puzzle	the largest Stanford integer benchmark, written in a traditional procedure-oriented style (170 lines long)
richards	a medium-sized operating system simulation benchmark (400 lines long)
pathCache	a short SELF program in frequent use today (140 lines long)
primMaker	a larger SELF program in occasional use today (1000 lines long)
parser	a medium-sized SELF program originally written as a programming exercise (400 lines long)

Section B.2 of Appendix B describes our measurement procedures in detail. Appendix C contains the raw data for all the measurements.

14.1.2 The Hardware

We did all our measurements on a Sun-4/260 workstation configured with 48MB of main memory. The Sun-4/260 workstation is based on the SPARC, a RISC-style microprocessor with hardware register windows and delayed branches and calls [HP90]. The implementation of the SPARC on the Sun-4/260 has a 62ns cycle time, 8 register windows, two-cycle loads (if the target register is not used in the following instruction, three cycles otherwise), and three-cycle stores, assuming cache hits. The SPARC also provides limited hardware support for tagged arithmetic, although none of the systems measured, including the SELF system, currently exploit this hardware.

14.1.3 Charting Technique

In all charts, bigger bars are better in the sense that they correspond to a more efficient implementation. Execution and compilation performance is reported in terms of speed, with taller bars corresponding to faster systems. Compiled code space costs are reported in terms of density (the inverse of space usage), with taller bars corresponding to more space-efficient systems (systems requiring less space for compiled code). Compiled code space numbers include only space for machine instructions and exclude space costs for debugging information and the like. Section B.6 of Appendix B reports measurements of these additional space costs for the SELF system.

14.2 Performance versus Other Languages

To determine the overall effectiveness of the new techniques described in this dissertation, we compared the performance of the SELF implementation to implementations of several other languages.

- We want to compare our SELF implementation to that of a traditional optimized language implementation. This comparison will tell us how well our techniques do in narrowing the gap in performance that previously existed between pure object-oriented languages and traditional optimized languages. Also, the performance of a traditional optimized language places something of an upper bound on the performance we can reasonably expect from our SELF implementation. We measured the optimizing C compiler provided standard with SunOS 4.0.3 UNIX, invoked with the **`-O2`** flag. The C version of the **`richards`** benchmark really is written in C++ (version 1.2 of AT&T's **`cfront`** C++-to-C translator) so that the **`runTask`** message can be implemented by a C++ virtual function call.
- We also want to compare our SELF implementation to the performance of the best existing implementation of a pure object-oriented language, to compare the effectiveness of our techniques to previous techniques for implementing pure object-oriented languages. We measured the performance of ParcPlace Smalltalk-80, version V2.4 β2, the fastest commercial implementation of Smalltalk. ParcPlace's Smalltalk-80 implementation includes the Deutsch-Schiffman techniques for constructing a fast Smalltalk implementation, described in section 3.1.2. Unfortunately, only execution speed results are available for Smalltalk; neither compilation speed nor compiled code space usage could be measured.
- Finally, we also are interested in the relative performance of our SELF system and some sort of Lisp-based system. Lisps provide many of the same features as does SELF, such as support for generic arithmetic, closures, and dynamic type-checking. Unlike SELF, however, Lisps also provide direct procedure calls, direct variable accesses, and built-in control structures. Comparing SELF to a Lisp system can help determine how well SELF handles generic arithmetic compared to existing techniques in Lisp systems and how well SELF optimizes away the extra overhead of message passing and completely user-defined control structures. We measured the ORBIT compiler (version 3.1) for T, a dialect of Scheme, described in section 3.3.2. This system is widely regarded as a high-quality Lisp implementation that supposedly competes well against traditional language implementations.

  Many Lisps include special low-level primitive operations that avoid the overhead of the corresponding general, safe primitive operations. For example, T includes the **`fx+`** operation which assumes its arguments are *fixnums* (Lisp jargon for fixed-precision integers that fit in a machine word, equivalent to **`SmallInteger`** objects in Smalltalk and **`int`** in C). **`fx+`** is more efficient than the generic **+** operation since **`fx+`** assumes its arguments are fixnums and can be compiled down to a single machine **`add`** instruction, much like the **+** operator in C. However, **`fx+`** does *not* verify the assumption that its arguments are in fact fixnums before it adds them, nor does it check for overflows after the addition; consequently, **`fx+`** is unsafe. This distinction between slow, safe operations and fast, unsafe operations forces programmers to choose explicitly where to use normal generic arithmetic (and be willing to pay the cost of the better semantics) and where to use the faster **`fx+`**-style arithmetic (and be willing to take responsibility for the unverified assumptions).

  Lisps also typically include directives that allow the programmer to invoke additional optimizations such as inlining. T includes a **`define-integrable`** form that works just like **`define`** (i.e., binding a name to a value or function) except that the compiler will inline the bound value or function whenever the name is evaluated. In T, inlining is an optimization that must be invoked explicitly by programmers.

  We expect that most T programmers normally would use **+** and **`define`**. Sometimes, however, T programmers may feel the need to use unsafe operations and explicit directives to get better performance. In fact, nearly all benchmark performance results for Lisps are reported after such hand optimizations, with **+** replaced with **`fx+`** where possible and with **`define-integrable`** sprinkled in where desired. To capture these two styles of usage, we wrote *two* T versions of each benchmark: one as we would expect a normal programmer to have written the benchmark (and thus measuring the expected performance of T for the average programmer), and another hand-optimized version of the benchmark using **`fx+`** and **`define-integrable`** more like what a benchmarker would measure (and thus measuring the best possible performance of T as seen by the expert).

  The two versions of the **`richards`** benchmark use T's structure facility to implement task objects, and declare **`runTask`** as a T operation, with different methods invoked for different types of task objects.

The following table summarizes the language implementations we measured:

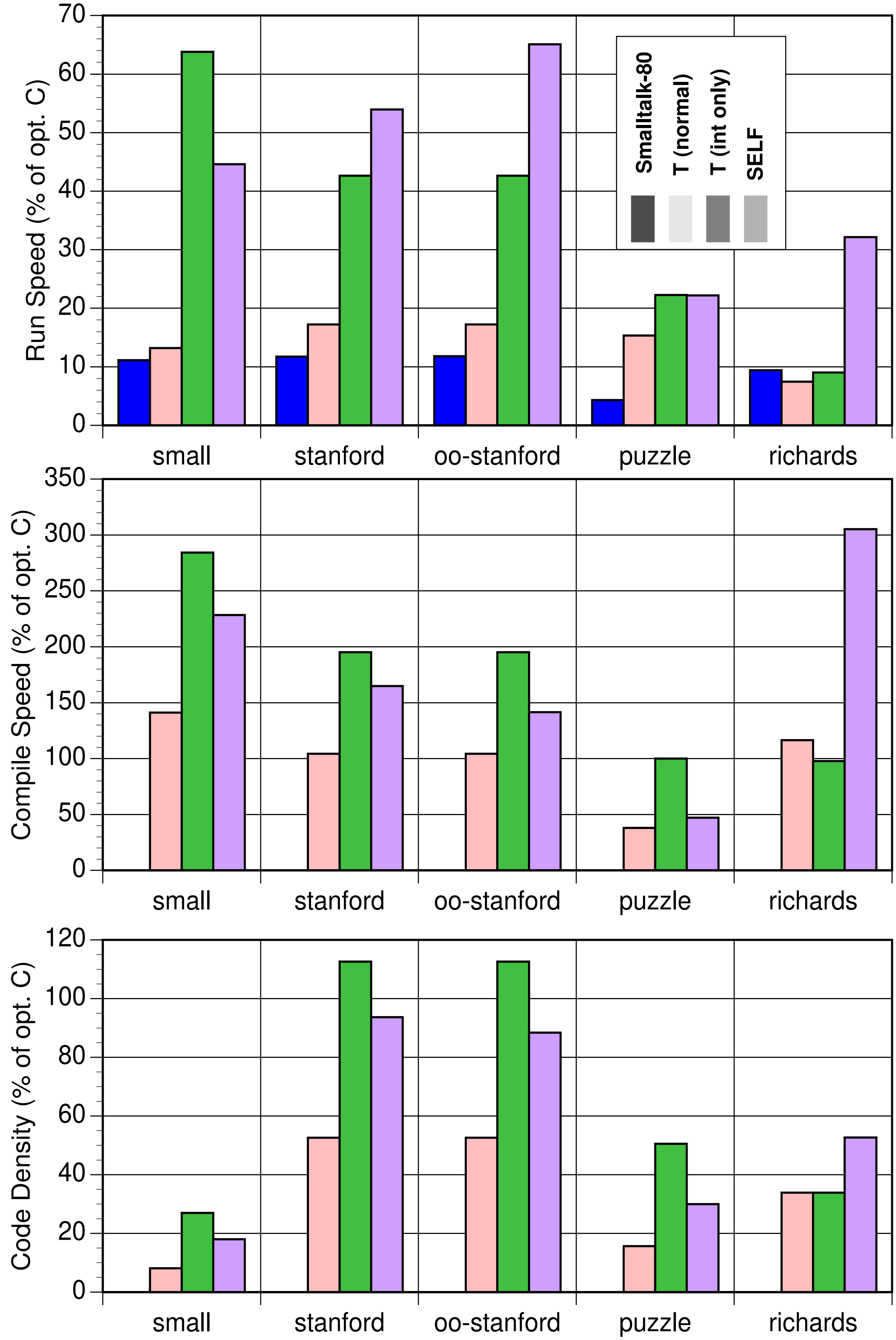
C	SunOS 4.0.3 standard optimizing C compiler (**`-O2`**)
Smalltalk-80	ParcPlace Systems Smalltalk-80 V2.4 β2, the fastest commercial Smalltalk implementation
T (normal)	ORBIT compiler for T (version 3.1), "normal" coding style
T (int only)	ORBIT compiler for T (version 3.1), integer-specific hand-optimized coding style
SELF	SELF implementation

The charts on the next page compare the execution speed, compilation speed, and compiled code space efficiency of these systems on the benchmarks (compilation speed and compiled code space efficiency results are unavailable for Smalltalk-80). All results are reported as a percentage of the results for optimized C.

These SELF benchmark suites and programs run between 22% and 65% of the speed of optimized C. This level of performance is about 5 times better than the performance of ParcPlace Smalltalk-80, despite the fact that SELF actually is harder to compile efficiently than Smalltalk (primarily because SELF accesses variables via messages while Smalltalk accesses them directly). Perhaps surprisingly, SELF runs between 1.4 and 4 times faster than T compiled by the ORBIT compiler, when running "normal" T programs, even though the T benchmarks use only direct procedure calls (excluding the one operation invocation site in **`richards`**) and only built-in control structures. SELF outperforms T in most cases even when comparing against fixnum-specific hand-optimized T programs. These comparisons provide evidence that the SELF compiler is doing an excellent job of eliminating the run-time overhead associated with message passing, user-defined control structures, and generic arithmetic.

The compilation speed of the SELF implementation is quite reasonable when compared to other optimizing language implementations. SELF's compilation speed is better than optimized C's for all cases except **`puzzle`**; SELF compiles **`richards`** 3 times faster than C++. SELF compiles faster than the ORBIT compiler on the "normal" T benchmarks; ORBIT usually compiles the integer-specific version of the T benchmarks faster than the SELF compiler. We could not measure the compilation speed for ParcPlace Smalltalk-80, but we believe that it is significantly faster than any of the other implementations, principally because the Smalltalk-80 compiler does virtually no optimization during compilation. Unfortunately, the performance of the SELF compiler is still too slow to go unnoticed by the SELF programmer, as was one of our original goals for the project. Speeding the compiler further without sacrificing execution performance continues to be an active area of research.

SELF is not as space-efficient as optimized C. SELF uses about 6 times as much space as C for the micro-benchmarks and about 3 times as much space for the **`puzzle`** benchmark. However, for the Stanford integer benchmarks, SELF incurs a space overhead of less than 20% compared to C, and for **`richards`** (the largest, most data-structure-oriented of these benchmarks), SELF consumes only twice as much space as optimized C++. These relatively low space overheads for the majority of the benchmarks are remarkable, considering that many of the SELF compiler's techniques such as splitting trade away compiled code space to get faster execution times. Given the falling costs of memory, the SELF system's extra space requirements seem quite reasonable. SELF uses less compiled code space than normal T programs compiled by the ORBIT compiler, and not much more space than integer-specific T programs. These results confirm the practicality of our compilation techniques in terms of space costs.



Run Speed (% of opt. C)
70
60
50
40
30
20
10
0
Smalltalk-80
T (normal)
T (int only)
SELF
small
stanford
oo-stanford
puzzle
richards
Compile Speed (% of opt. C)
350
300
250
200
150
100
50
0
small
stanford
oo-stanford
puzzle
richards
Code Density (% of opt. C)
120
100
80
60
40
20
0
small
stanford
oo-stanford
puzzle
richards

Language implementations typically trade off execution speed and compilation speed against one another: the compiler usually has to work harder to produce code that runs faster. The practicality of a particular language depends in large part on how well its implementation balances these two competing goals. The chart below summarizes the performance of the language implementations by scatter-plotting the execution and compilation speed results for each system, averaging together all the benchmarks. We assigned the Smalltalk-80 implementation a compilation speed 4 times faster than the optimizing C compiler. This figure is only a guess, intended to roughly illustrate the Smalltalk-80 implementation's position in the chart; the execution speed figure is accurate, however.

paste chart-page-164.ps here

SELF performs better in execution speed than either version of the T/ORBIT system, and compiles faster than ORBIT on the normal version of the T benchmarks. SELF runs these benchmarks at an average of 50% the speed of optimized C and with faster compilation speed than the optimizing C compiler.

14.3 Relative Effectiveness of the Techniques

Now that we understand the overall effect of the new techniques, we can explore in more detail the relative effectiveness, and accompanying costs, of each of the new techniques developed as part of the SELF compiler. The results will help identify those techniques that are worth including in any future implementation of a language like SELF.

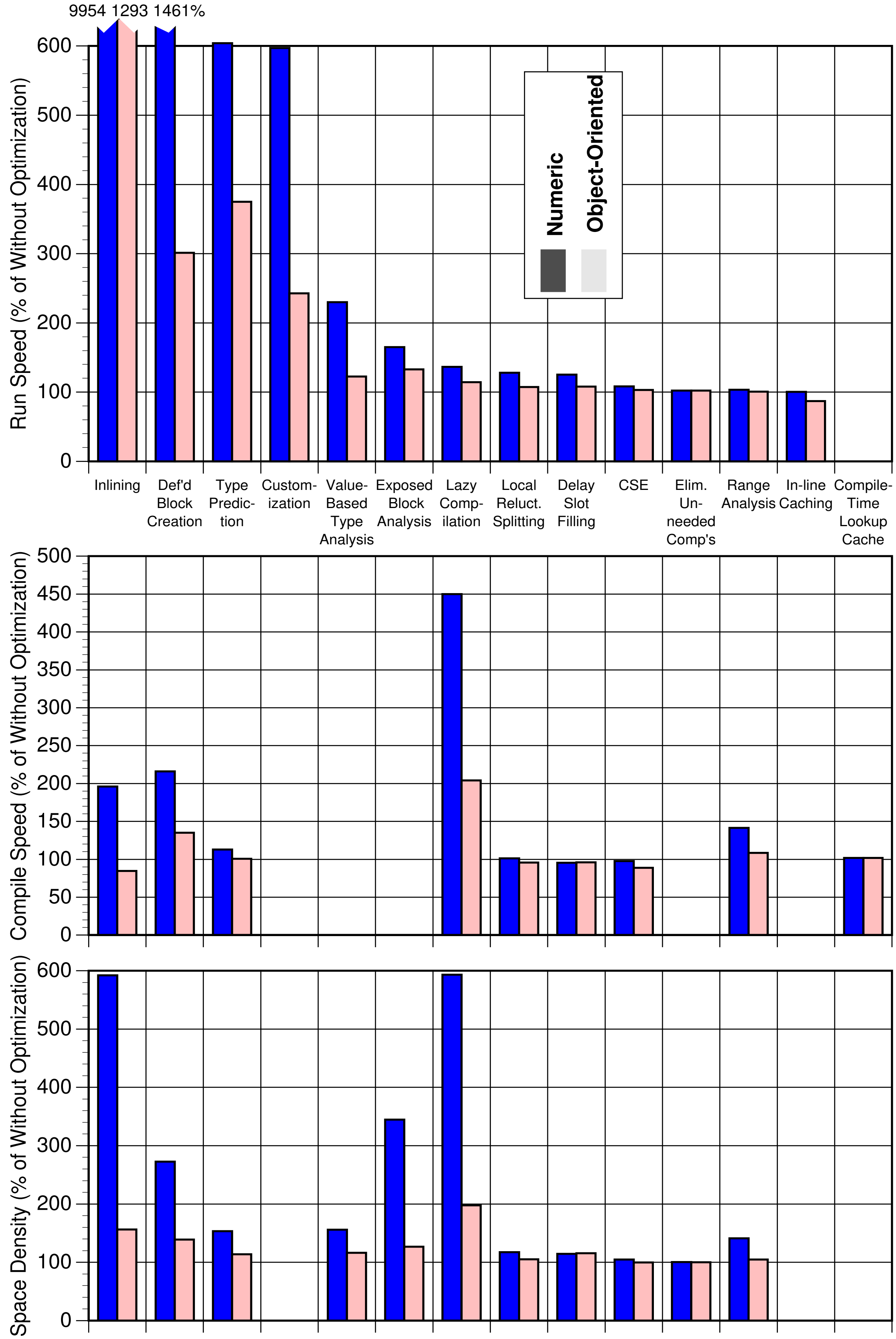
We calculate the benefits and costs of each technique by comparing the performance of the standard SELF configuration to the performance of a configuration with the technique disabled. In the charts, the effect of an optimization is shown by reporting the performance of the normal SELF system including the optimization relative to the performance of the SELF system with the optimization disabled; this approach remains consistent with our general visual theme of bigger bars indicating better results, in this case bigger bars indicating a more important or effective optimization.

The charts on the following page summarize the effect on performance of the various optimizations implemented in the SELF compiler. Detailed information for each optimization may be found in sections B.3 and B.4 of Appendix B.

Four optimizations are clear winners: inlining, deferred block creations, type prediction, and customization. Type analysis presumably is also a winner, but its impact could not be directly measured. Value-based type analysis, exposed block analysis, lazy compilation, local reluctant splitting, and delay slot filling also make significant contributions; lazy compilation in particular makes a huge improvement in compilation speed and compiled code space utilization. Of course, register allocation is also very important, but the impact of the particular details of the SELF register allocator, such as allocating variables instead of names, were not measured.

Common subexpression elimination, eliminating unneeded computations, range analysis, in-line caching, and caching compile-time lookups do not appear particularly important. Range analysis improves compilation speed and compiled code space density, therefore redeeming its rather slim execution speed improvement. As described in section B.3.4, in-line caching would appear more effective for the larger, more object-oriented benchmarks if not for a performance bug in the run-time system that halves the performance of the **`pathCache`** benchmark. Common subexpression elimination provides significant improvements for some benchmarks such as **`oo-bubble`** (see Appendix C), but overall performance improves by less than 5%.

In nearly all cases, the optimizations made a bigger difference for the smaller, more numeric benchmarks than for the larger, more object-oriented benchmarks. Part of this disparity probably stems from our concentration on the smaller benchmarks as the SELF compiler was being designed and implemented. Had we concentrated more on the object-oriented benchmarks, the trend might have been the reverse. Since both styles of program are important, we view this difference in effectiveness as an opportunity for future work.



9954 1293 1461%
Run Speed (% of Without Optimization)
600
500
400
300
200
100
0
Numeric
Object-Oriented
Inlining
Def'd Block Creation
Type Prediction
Customization
Value-Based Type Analysis
Exposed Block Analysis
Lazy Compilation
Local Reluct. Splitting
Delay Slot Filling
CSE
Elim. Unneeded Comp's
Range Analysis
In-line Caching
Compile-Time Lookup Cache
Compile Speed (% of Without Optimization)
500
450
400
350
300
250
200
150
100
50
0
Space Density (% of Without Optimization)
600
500
400
300
200
100
0

14.4 Some Remaining Sources of Overhead

The SELF system does not (yet) run as fast as a traditional language implementation such as optimized C. We would like to know the sources of this remaining gap in performance, to guide future work. In this section, we examine many of the sources of overhead that traditionally have slowed down pure object-oriented languages, such as extra run-time type tests, overflow checks, and array bounds checks. We can use the results to see how well the new techniques applied to SELF reduce these traditional sources of overhead, and we can direct future research towards reducing the cost of any overhead that remains significant.

We measure the cost of a particular source of overhead by constructing a version of the system without the source of overhead and comparing the speed of this version to the standard version of the system. Frequently, this new version of the system is not a legal SELF implementation in that not all SELF programs will run correctly on it, but for the benchmarks we measure, the altered system runs correctly. Unfortunately, not every possible source of overhead could be measured this way. For several key sources of overhead, it was too difficult to produce configurations that would simulate their absence. For example, the overhead of message passing, inheritance, dynamic typing, and user-defined control structures is nearly impossible to remove and measure directly; too much of the system internals and the benchmark source code relies on these features being present. Also, any overhead introduced as a result of deficiencies in the implementation of the SELF compiler cannot be measured in this way.

In the charts on the next page, we summarize the cost of those sources of overhead we were able to measure by displaying the performance of a configuration with the source of overhead removed relative to the performance of the standard SELF system; bigger bars mean that the overhead is more costly given SELF's current implementation technology. We also report the performance of two additional configurations:

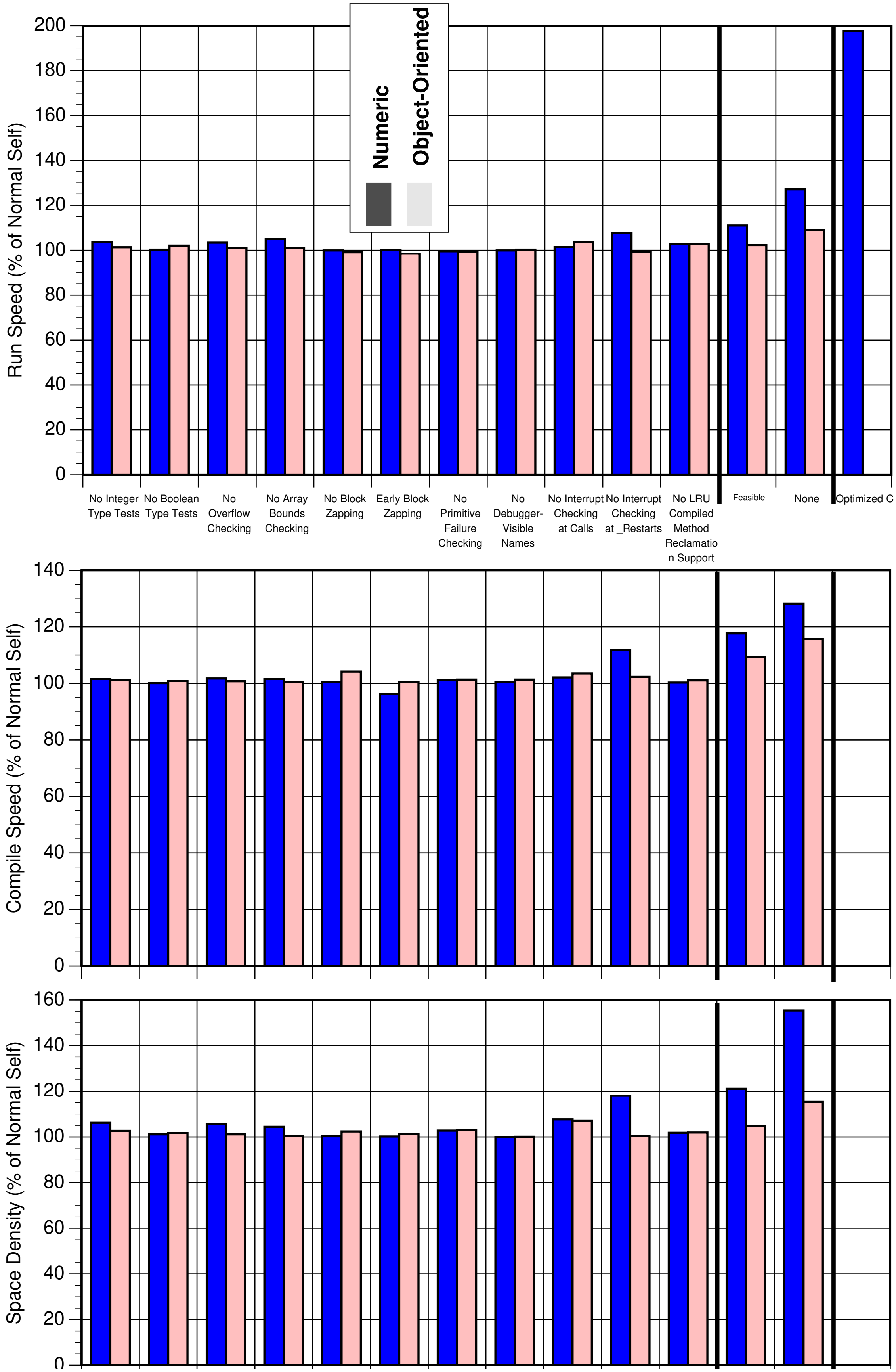
- **`feasible`** excludes those sources of overhead that might be avoided by implementing parts of the system differently: block zapping, interrupt checking via polling, and LRU compiled method reclamation support.
- **`none`** excludes all language and implementation overhead we measured, to try to produce a system that is as fast as possible.

We also report the execution speed of optimized C for comparison. More detailed analysis of each of the sources of overhead may be found in section B.5 of Appendix B.

Surprisingly, no single traditional source of overhead that we could measure imposes a significant execution speed cost. Array bounds checking, type testing, overflow checking, interrupt checking, and LRU compiled method reclamation support are the only measured sources of overhead with a non-trivial execution time cost, and none incurs more than 10% cost. This low cost reinforces the earlier overall performance measurements in illustrating how well the techniques implemented in the SELF compiler work at reducing or eliminating the sources of overhead that historically have plagued implementations of pure object-oriented languages.

There is still a fairly sizable gap in performance between the fastest version of SELF and optimized C that remains unaccounted for. Unless this gap stems solely from poorer implementation in the SELF compiler of traditional optimizations such as global register allocation or loop-invariant code motion, further improvements in the speed of pure object-oriented languages await additional insight.

Compilation time costs and compiled code space costs for the sources of overhead we can measure are fairly small. Polling-style interrupt checking and block zapping are the most compile-time- and compiled-code-space-consuming features to support. Interestingly, these worst offenders are all related to the particular system architecture of the SELF implementation rather than to any required language or environment features, so there is hope that this overhead could be reduced by redesigning various parts of the implementation.


Numeric
Object-Oriented
Run Speed (% of Normal Self)
200
180
160
140
120
100
80
60
40
20
0
No Integer Type Tests
No Boolean Type Tests
No Overflow Checking
No Array Bounds Checking
No Block Zapping
Early Block Zapping
No Primitive Failure Checking
No Debugger-Visible Names
No Interrupt Checking at Calls
No Interrupt Checking at _Restarts
No LRU Compiled Method Reclamatio n Support
Feasible
None
Optimized C
Compile Speed (% of Normal Self)
140
120
100
80
60
40
20
0
Space Density (% of Normal Self)
160
140
120
100
80
60
40
20
0

14.5 Summary

By and large, the new techniques we developed for efficiently implementing SELF work well: SELF runs about half as fast as optimized C for the benchmarks we measured, a five-fold improvement over the best previous implementation of a similar pure object-oriented language (ParcPlace Smalltalk-80). SELF's performance is more than double the speed of a well-respected implementation of a similarly dynamically-typed language supporting generic arithmetic but also direct procedure calls and built-in control structures (the ORBIT compiler for T). Compile time and compiled code space costs are comparable to these other language implementations, with the exception of ParcPlace Smalltalk-80 which we believe compiles much faster than the other language implementations.

The bulk of the SELF compiler's good performance can be attributed to a few optimizations. Certain techniques simply must be applied to have any hope of decent performance, including inlining and deferred block creations. Type prediction and customization each improve execution performance by about a factor of four or five. Value-based type analysis, exposed block analysis, splitting, lazy compilation of uncommon branches, and delay slot filling also make significant contributions to run-time performance. Other optimizations have more modest benefits; common subexpression elimination and integer subrange analysis in particular were fairly complex to implement and were expected to have greater pay-offs. Some important techniques could not be measured, such as the effectiveness of type analysis or the space costs of customization, since these techniques were too integral to the system to disable.

Measurements of the remaining cost of some traditional sources of overhead in implementations of pure object-oriented languages indicate that none of the traditional bottlenecks continues to incur a significant cost. This result confirms the effectiveness of the new techniques, but unfortunately does not provide much help in guiding future research down profitable avenues.