Appendix B Detailed Performance Evaluation

The first two sections of this appendix contain additional information about the benchmarks and the measurement procedures. This appendix also includes detailed analyses of the effectiveness of the various techniques included in the SELF implementation (in sections B.3 and B.4) and of some of the remaining sources of overhead that slow SELF down compared to optimized C (in section B.5). Finally, section B.6 discusses the space costs of the extra information generated by the SELF compiler beyond just native machine instructions.

B.1 Detailed Description of the Benchmarks

We measured the following micro-benchmarks:

- **`recur`** is a tiny recursive benchmark that stresses method call and integer comparison and subtraction, adapted from the **`testActivationReturn`** Smalltalk-80 micro-benchmark [Kra83].
- **`sumTo`** adds up all the numbers from its receiver (1) to its argument (10000), 100 times over. **`fastSumTo`** is the same as **`sumTo`**, except that the body of **`sumTo`** is inlined into the outer loop (manually in the C, Smalltalk, and T versions, automatically in the SELF version). **`sumFromTo`** is similar to **`sumTo`**, except that it adds up all the numbers from its first argument (1) to its second (10000), initializing the accumulator to its receiver (0). These benchmarks stress generic arithmetic applied to integers and user-defined control structures. **`sumFromTo`** is a benchmark used to measure the TS Typed Smalltalk compiler (described in section 3.1.3).
- **`nestedLoop`** increments a counter inside a doubly-nested **`for`**-style loop, each loop iterating 100 times; this test is itself run 100 times. This test also stresses generic arithmetic applied to integers and user-defined control structures, and is another benchmark used to measure the TS Typed Smalltalk compiler.
- **`atAllPut`** stores an integer (7) into all elements of a 100,000-element-long vector, and so stresses iterating through and storing into arrays. It was originally suggested to us by Peter Deutsch as an interesting micro-benchmark [Deu89].
- **`sumAll`** adds up all the elements of a 100,000-element-long vector. This test stresses iterating through arrays and generic arithmetic of integers, and is another benchmark used on the TS Typed Smalltalk compiler.
- **`incrementAll`** increments all the elements of a 100,000-element-long vector, stressing iterating through arrays, storing into arrays, and generic arithmetic on integers. This benchmark is another TS Typed Smalltalk benchmark.
- **`sieve`** finds all the primes between 1 and 8190 using Eratosthenes' sieve algorithm and stresses integer calculations, integer comparisons, and accessing arrays of booleans.
- **`tak`** executes the recursive Tak benchmark from the Gabriel Lisp benchmarks [Gab85], which stresses method calling and integer arithmetic. **`takl`** performs the same algorithm, but uses lists of cons-cells to represent integers, and so additionally stresses list traversals and memory allocation.

Most of these micro-benchmarks are 1 to 3 lines long. **`sieve`** is 8 lines long, **`tak`** is 6 lines long, and **`takl`** is 10 lines long.

We measured the following Stanford integer benchmarks, each of which exercises integer calculations, generic arithmetic, array accessing, and user-defined control structures (particularly **`for`**-style loops):

- **`perm`** and **`oo-perm`** are recursive permutation programs (25 lines long each).
- **`towers`** and **`oo-towers`** recursively solve the Towers of Hanoi problem for 14 disks (60 lines long each).
- **`queens`** and **`oo-queens`** solve the eight-queens placement problem 50 times (35 lines long each).
- **`intmm`** and **`oo-intmm`** multiply two 40-by-40 matrices of random integers. Since two-dimensional matrices are not supported directly by SELF, Smalltalk, or T, in these languages a matrix is represented by an array of arrays, in contrast to the contiguous representation used in the C version. These benchmarks are each 30 lines long.
- **`quick`** and **`oo-quick`** sort an array of 5000 random integers using the quicksort algorithm (35 lines long each).
- **`bubble`** and **`oo-bubble`** sort an array of 500 random integers using the bubblesort algorithm (20 lines long each).

- **`tree`** and **`oo-tree`** sort 5000 random integers using insertion into a sorted binary tree data structure (25 lines long each). These benchmarks stress memory allocation and data structure manipulation instead of array accessing.
- **`puzzle`** solves a time-consuming placement problem (170 lines long). This benchmark is unusual in that its source code size is dominated by code to initialize the puzzle data structures, and so demands good compiler speed more than the other benchmarks.

Source code for all benchmarks is available from the author upon request.

B.2 Measurement Procedures

We measured the speed of the SELF, C, and T implementations in milliseconds of CPU time (user plus system time), as reported by the UNIX **`getrusage`** system call for SELF and C and the **`time`** function for T. The Smalltalk-80 implementation only reports elapsed real time, so all measurements of the performance of Smalltalk programs are in milliseconds of real time rather than CPU time. Since the Smalltalk-80 real-time numbers did not fluctuate significantly from run to run (indicating that any extra overhead included in real time for Smalltalk-80 was relatively constant), and the real-time measurements for SELF and for C were about the same as the corresponding CPU-time measurements (indicating that any extra overhead included in real time for SELF and C was small), we believe that the CPU time for Smalltalk-80 is likely to be close to the real-time measurements. Therefore, we feel that comparisons among Smalltalk-80's real-time measurements and the other language's CPU-time measurements are reasonably valid.

We measured the run time of each benchmark by executing the benchmark in a loop of 10 iterations, reading the elapsed time before and after the loop, and dividing the resulting difference by 10. This measurement of 10 iterations helps to increase the effective resolution of the hardware clock (which is only to the nearest 10ms on a Sun-4/260) and smooth over any variations from run to run.

The compile times for the C version of each benchmark were calculated by using the UNIX **`time`** command around the compilation and optimization of a source file containing only the benchmark being measured. Thus, compilation time for C includes the time for reading and writing files but not the time for linking the output object file into an executable program. The compile times for the ORBIT compiler were computed similarly by using T's **`time`** function around the invocation of the **`orbit`** function on the name of the file containing the benchmark. Thus, T's compile time includes the time to read and write files but not the time to load the resulting output file into the running T system.

Separating compile time from run time in SELF and Smalltalk is complicated by dynamic compilation. The first execution of a piece of code includes both execution time and compilation time. Our technique for calculating compilation and execution time for a SELF benchmark is first to flush the compiled code cache (to make sure none of the code the benchmark will use is already compiled in the cache). Then the benchmark is run once; this first run includes both compilation time and run time. Then the benchmark is run 10 more times; these runs include only execution, since the compiled code is already in the compiled code cache thanks to the first run. The final execution time is calculated as the time of the last 10 runs divided by 10 (just as for C and T), and the compilation time is calculated as the time for the first run minus the average execution time for the last 10 runs. This approach assumes that the execution time of the first run of the benchmark is equal to the average subsequent execution, which may not be completely accurate (e.g., because of hardware caching and paging effects), but is probably close enough to use the calculated compile times as a rough measure of the actual compile-time overhead of our implementation. Unfortunately, the Smalltalk-80 system does not provide a mechanism to flush the compiled code cache, so compilation speed numbers are unreliable. Consequently, the same process for measuring SELF is performed for Smalltalk (ignoring the cache flush step), but only the average execution time numbers are retained. Compiled code space figures also are available for Smalltalk.

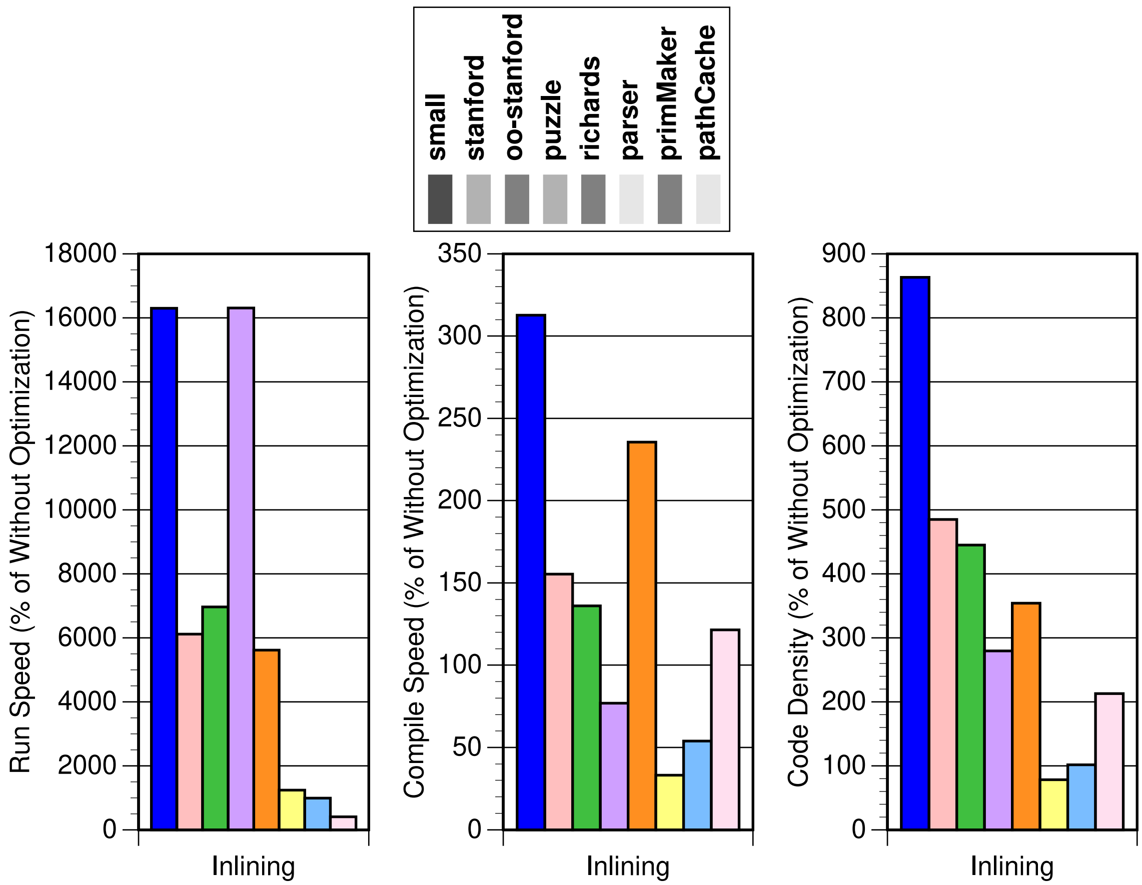
Frequently, to reduce the number of data points displayed in the charts, the results of individual benchmarks have been combined together to form a result for a suite of similar benchmarks. The average result for a benchmark suite was calculated by taking each individual benchmark's result, normalizing it to some reference result (typically either the performance of optimized C or the performance of the standard SELF configuration), and then taking the geometric mean of the normalized results for the benchmarks in the suite. Appendix C contains the original raw data for all the measurements.

B.3 Relative Effectiveness of the Techniques

This section explores the effectiveness of individual optimizations in detail, with the exception of splitting strategies which are the subject of the next section. These techniques will be covered in the order in which they were described in this dissertation. Summary information may be found in section 14.3.

B.3.1 Inlining

The SELF compiler relies on aggressive inlining to achieve good performance, as described in Chapter 7. To verify that the SELF system would be intolerably slow without inlining, we measured a version of SELF in which inlining of both messages and calls to primitives was disabled. The results are displayed in the following charts.

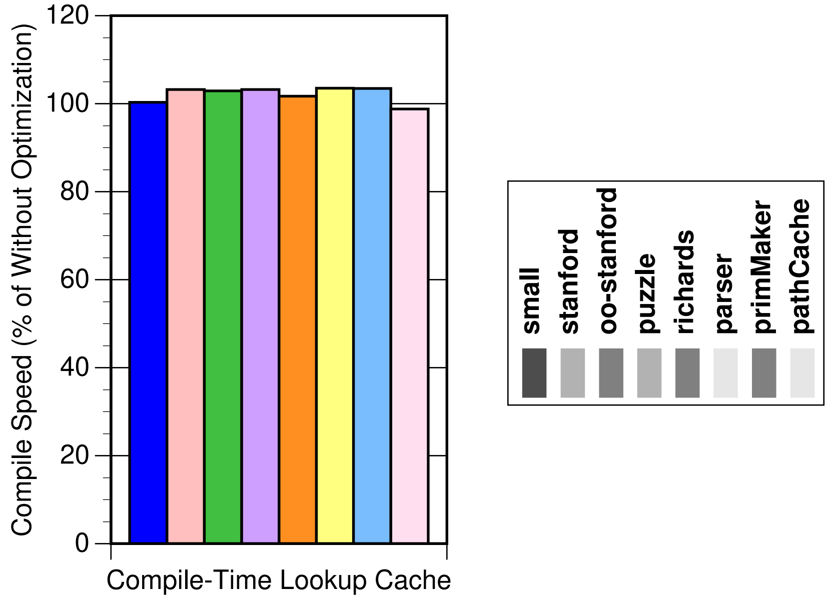


Inlining makes a huge difference in performance. The smaller numeric benchmarks run about two orders of magnitude faster with inlining, and even the larger non-numeric benchmarks run between 4 and 55 times faster with inlining. These results support our contention that aggressive inlining is the key to achieving good performance in pure object-oriented languages.

Surprisingly, inlining frequently speeds compilation and almost always saves compiled code space! This counter-intuitive result illustrates the difference between inlining in a traditional language and inlining in the SELF system. In SELF, the compiler uses inlining mostly for optimizing user-defined control structures and variable accesses, where the resulting inlined control flow graph is usually much smaller than the original un-inlined graph. These sorts of inlined constructs are already "inlined" in the traditional language environment. Inlining of larger "user-level" methods or procedures does usually increase compile time and compiled code space as has been observed in traditional environments; the SELF compiler simply spends much more of its time inlining things that shrink the control flow graph than things that expand it. Of course, a system never designed to perform inlining in the first place could compile faster than a compiler that could perform inlining but did not, so these results probably overstate the benefits of inlining for compilation speed.

B.3.2 Caching Compile-Time Message Lookups

The SELF compiler includes a message lookup cache to speed compile-time message lookups, as described in section 7.1.3. The following chart reports the effectiveness of this cache in speeding compilation.

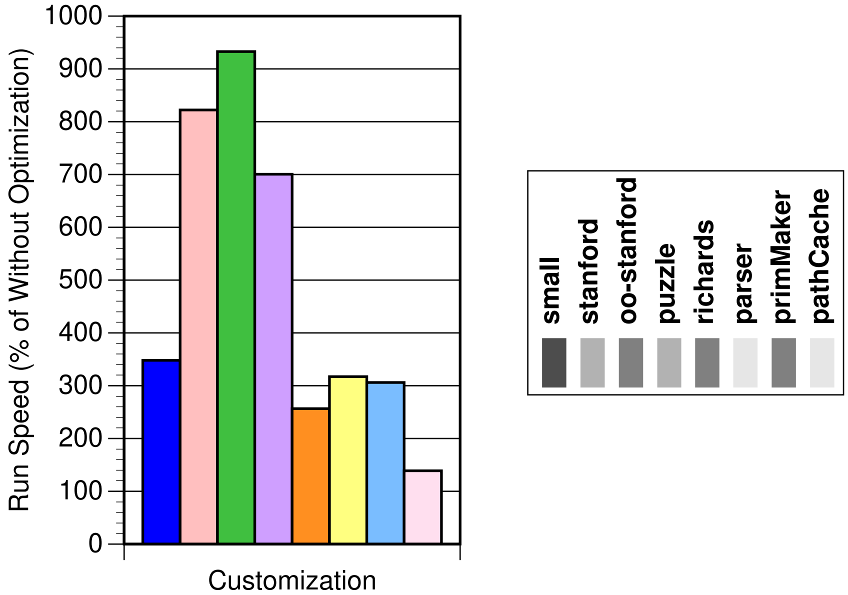


The compile-time message lookup cache speeds compilation by only a few percent. This poor showing is the result of being able to use the cache only within a single compilation and not across calls to the compiler. We are investigating system designs that would support a longer-lived compile-time message lookup cache which we believe could save a significant amount of compile time.

B.3.3 Customization

The SELF compiler uses customization to provide extra type information about **`self`** that enables much more inlining, as described in Chapter 8. Much of the implementation of the SELF system assumes that compiled methods are customized, so it is difficult to completely disable customization in order to measure its impact on the performance of the system. Fortunately, one aspect of customization can be disabled relatively easily: its contribution of the type of **`self`**. After the method prologue, the disabled compiler "forgets" the type of **`self`** by rebinding it to the unknown type. The run-time performance of this configuration should closely approximate the run-time performance of a version of the system with no customization at all, since the dominating effect of customization is this extra type information, not the duplication of methods and in-line caches. However, the compilation speed and compiled code space efficiency of the disabled configuration probably would be worse than a version of the system that did not customize at all, since the disabled configuration still can compile multiple versions of source methods, and so we cannot report accurate compile time costs and compiled code space costs for customization with this configuration.

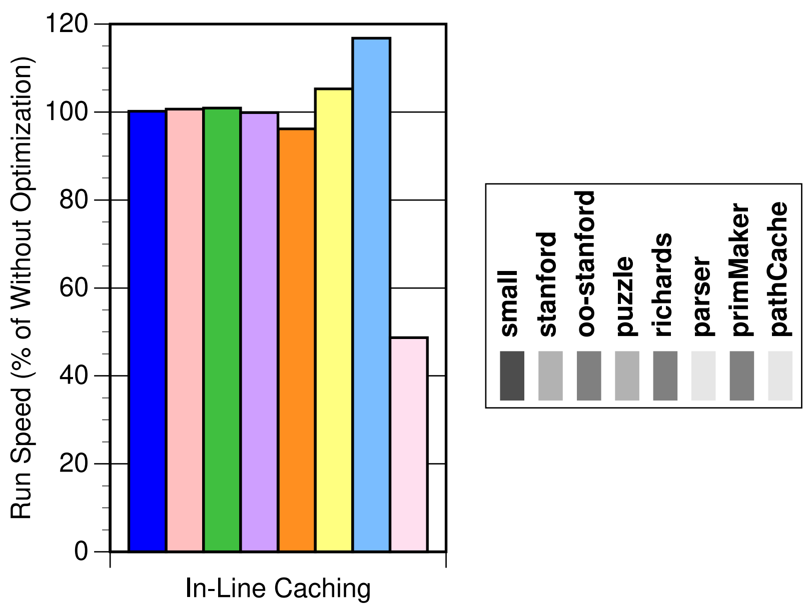
The following chart presents customization's impact on run-time execution performance.



Customization improves performance significantly, enabling these SELF benchmarks to run between 1.4 and 9 times faster. The extra type information provided by customization has been put to good use by the compiler in speeding run-time performance.

B.3.4 In-Line Caching

In-line caching speeds message sends that are neither inlined nor statically bound, as described in section 8.5. Since the SELF compiler is so effective at inlining messages, some question might remain as to whether in-line caching is still important for good performance. The following chart reports the impact of in-line caching on the execution performance of the SELF system; in-line caching affects neither compilation speed nor compiled code density.



In-line caching makes little impact on the smaller benchmarks, since most sends are inlined in these benchmarks and those that remain are frequently statically-bound to a single target method and so require no in-line caching. The **`primMaker`** and the **`parser`** benchmarks benefit more from in-line caching, but not by more than 20%. This result is rather surprising, given in-line caching's importance in the Deutsch-Schiffman Smalltalk-80 system, and it serves as confirmation of the effectiveness of inlining and static binding.

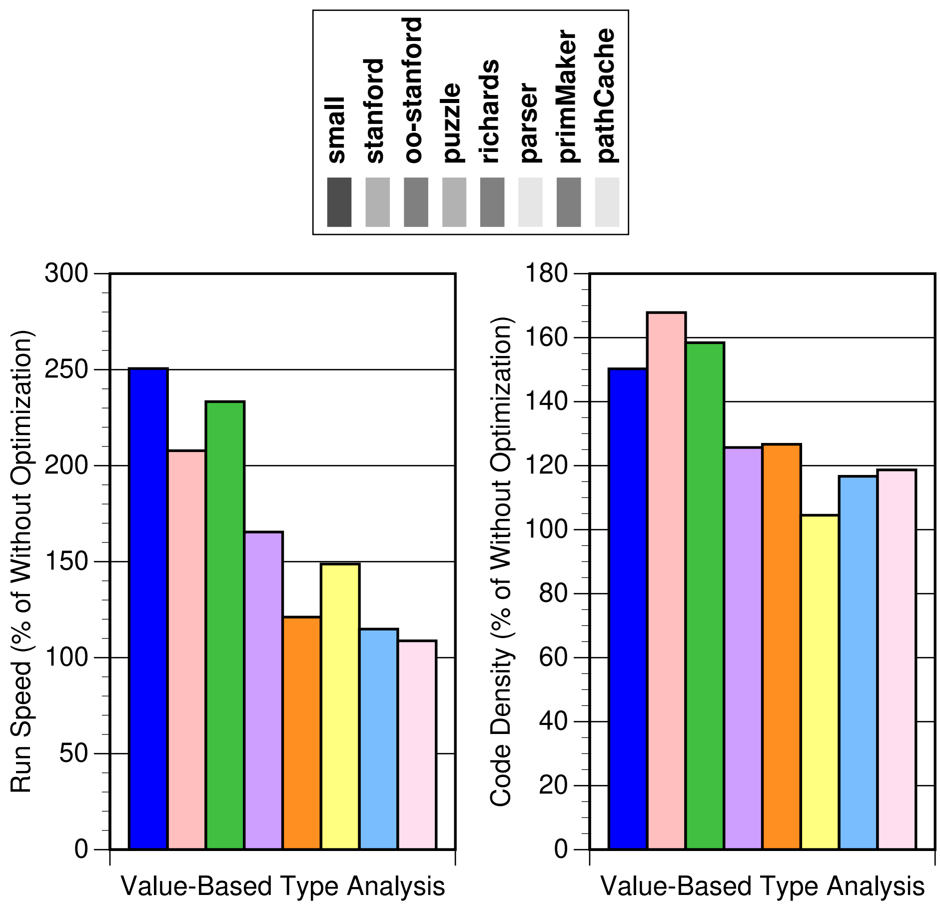
Amazingly, the **`pathCache`** benchmark runs more than twice as *slowly* with in-line caching! This unexpected result is caused by a performance bug in the run-time system (not the compiler) which slows down messages with more than 9 different receiver types. Since these measurements were made and the problem discovered, the implementation of in-line caches with many receiver types has been improved, and now in-line cached sends reportedly are almost always faster than uncached sends [Höl91].

B.3.5 Type Analysis

The SELF compiler relies on type analysis to propagate type information through the control flow graph and thus inline away more messages and avoid unnecessary type tests. Type analysis was the subject of Chapter 9 and much of Chapter 11. Like customization, type analysis is difficult to disable in the SELF compiler, since type analysis permeates the compiler's entire design. Therefore, we could not directly measure the effectiveness of type analysis using a version of SELF that performs no type analysis.

B.3.6 Value-Based Type Analysis

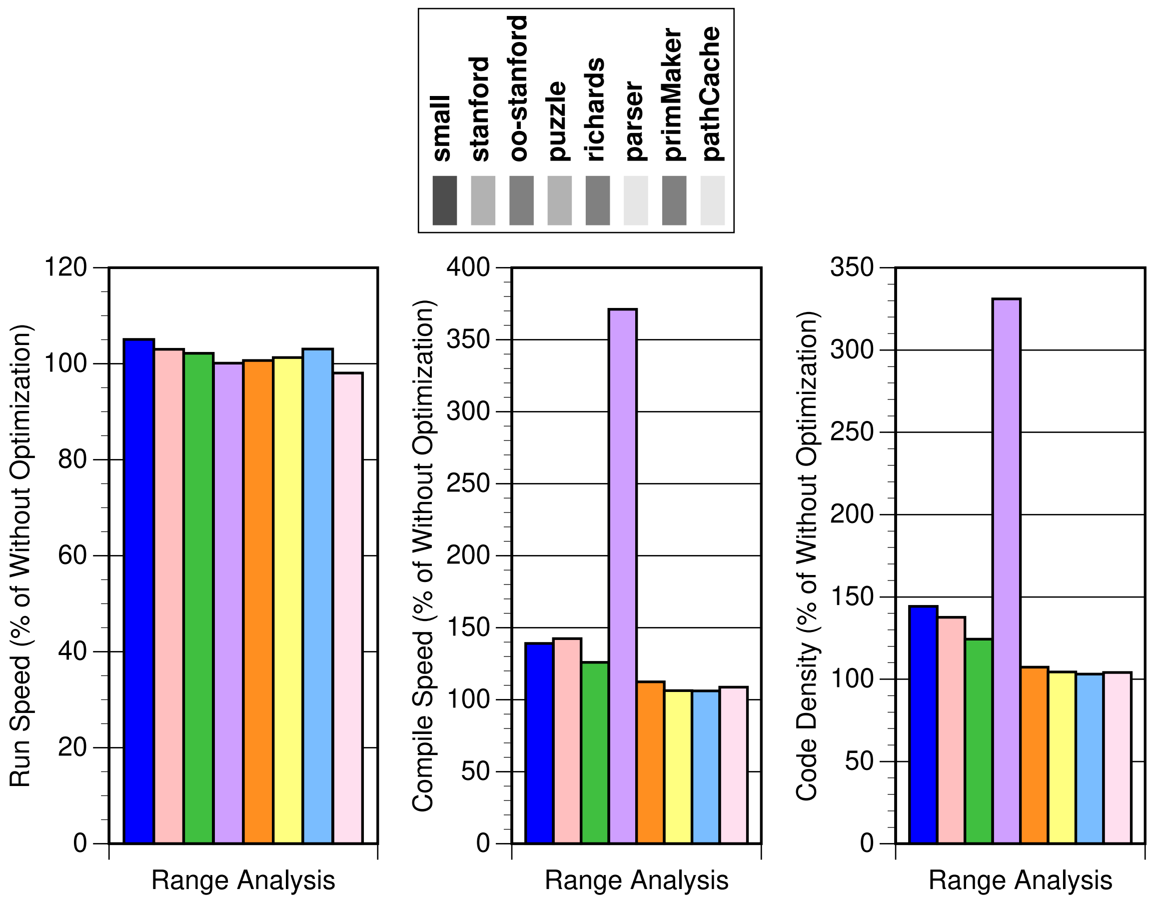
Fortunately, we can measure the effect of some parts of type analysis, such as the SELF compiler's use of value objects to link names to types and so improve the effectiveness of optimizations such as type prediction, as described in section 9.1.3. We can approximate a version of SELF without values by creating a new dummy value object whenever a name is assigned. Consequently, no two names will ever share the same value. Execution speed and compiled code space efficiency of this hobbled configuration should be close to a version of SELF without values; compilation speed would not since a compiler written without values should run faster than one written with values that have been rendered useless. The following charts report the impact of value-based type analysis on execution speed and compiled code space density.



These results show that value-based type analysis makes a significant improvement in execution performance. The smaller benchmark suites run more than twice as fast with value-based type analysis. The use of values improves the smaller, numerical benchmarks more than the larger benchmarks for two reasons. First, the numerical benchmarks rely more on type prediction for good performance (corroborated by the results in section B.3.8), and values improve the effectiveness of type prediction. Second, the smaller benchmarks send more messages to a single expression, for example an accumulator which receives a **+** message every time through a loop, and values allow the type information propagated from one name to another to be exploited when optimizing these sequences of messages. Value-based type analysis also improves the density of the compiled code by eliminating the need for repeated, redundant type tests which would take up additional compiled code space.

B.3.7 Integer Subrange Analysis

Integer subrange types support optimizations that depend on range analysis, such as eliminating unnecessary overflow checks and array bounds checks, as described in section 9.1.4.3. The effect of integer subrange types can be determined easily by using the more general integer map type wherever an integer subrange type would have been introduced. The charts below report the effectiveness of integer subrange analysis.

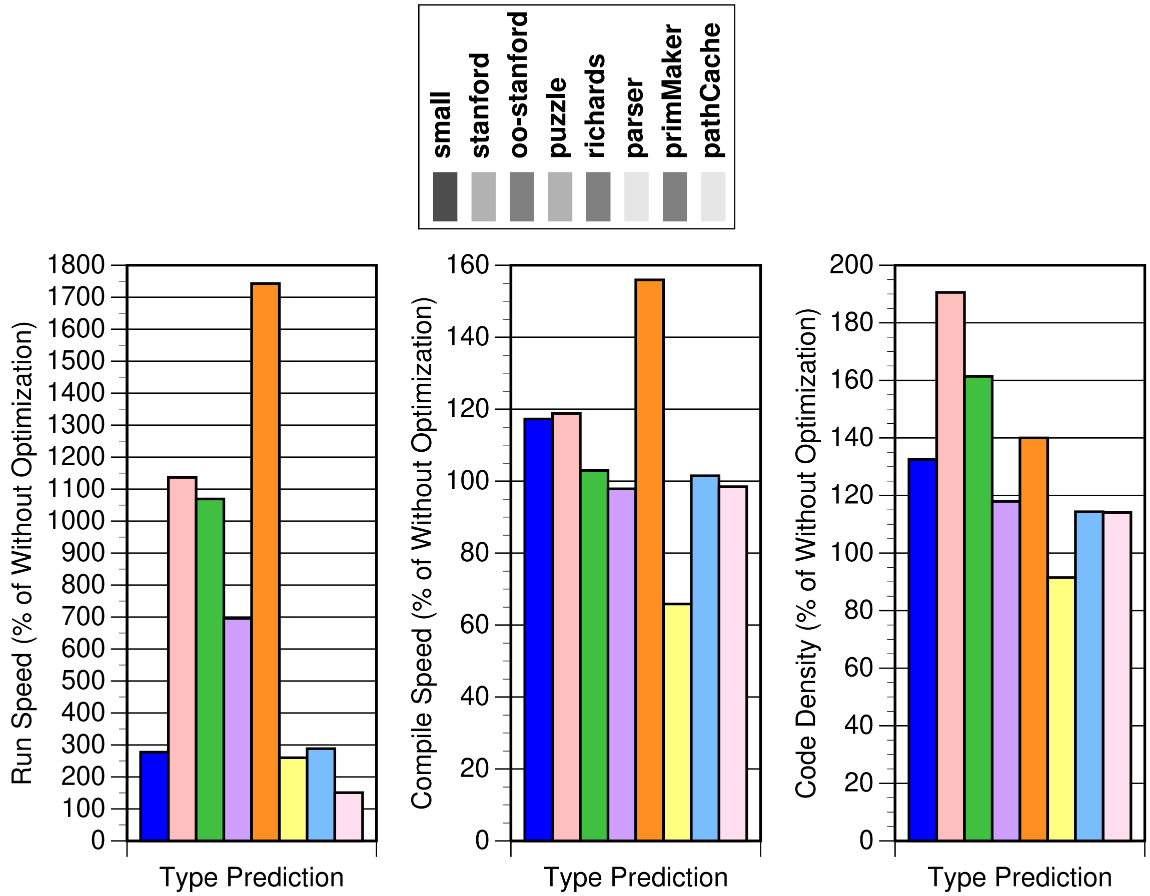


Integer subrange analysis seems to have little effect on execution performance, improving the speed of the benchmarks less than 5%. Range analysis' impact would be greater if not for two limiting factors. First, the current register allocator does not ensure that adjacent variables get allocated to the same location (as described in section 12.1.5), and so frequently a register move instruction remains after integer subrange analysis eliminates an overflow check. Second, range analysis currently is limited by requiring integer constants as upper and lower bounds. Allowing some form of symbolic bounds or constraints on unknown values might improve the effectiveness of range analysis, especially at eliminating array bounds checks for iteration over arrays of unknown size.

Integer subrange analysis frequently makes a large improvement in both compilation speed and compiled code space costs. Optimizing away possible uncommon branches via integer subrange analysis makes a big improvement in compilation speed, if not execution speed, and saves a lot of compiled code space.

B.3.8 Type Prediction

The SELF compiler uses type prediction (described in section 9.4) to increase the amount of type information available to the compiler for certain common message sends. The effectiveness of this technique can be easily measured by simply not performing type prediction. The following charts report type prediction's contribution to execution performance and its effect on compilation speed and compiled code density.



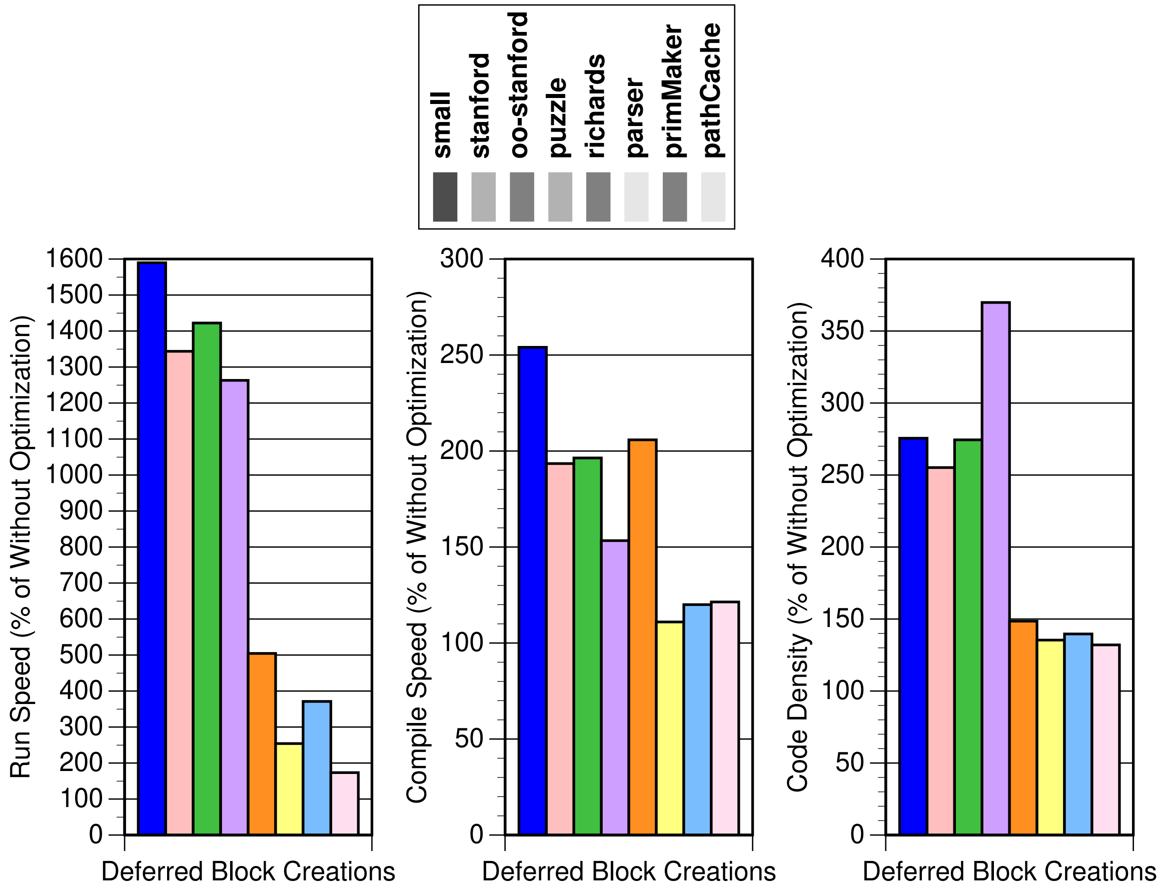
With type prediction SELF programs run much faster, between 1.5 and 17.5 times faster. Type prediction is at least as important as customization in achieving good performance on these benchmarks. Type prediction has a mixed effect on compilation speed, sometimes speeding and sometimes slowing compilation. The compiler sometimes can compile faster with type prediction because with lazy compilation of uncommon branches the common-case type-predicted version of a message send can be simpler and thus faster to compile than the original unpredicted message send. Type prediction reduces compiled code space costs for most benchmarks for much the same reason.

B.3.9 Block Analysis

Since blocks are such a key part of the SELF system, used by virtually all control structures, the SELF compiler incorporates several optimizations designed to reduce the cost of blocks. These optimizations were described in section 9.5.2.

B.3.9.1 Deferring Block Creations

The SELF compiler defers creating a block until it is first needed as a real run-time value, if at all. Disabling this optimization is easy. The charts below report the effect of deferring block creations.

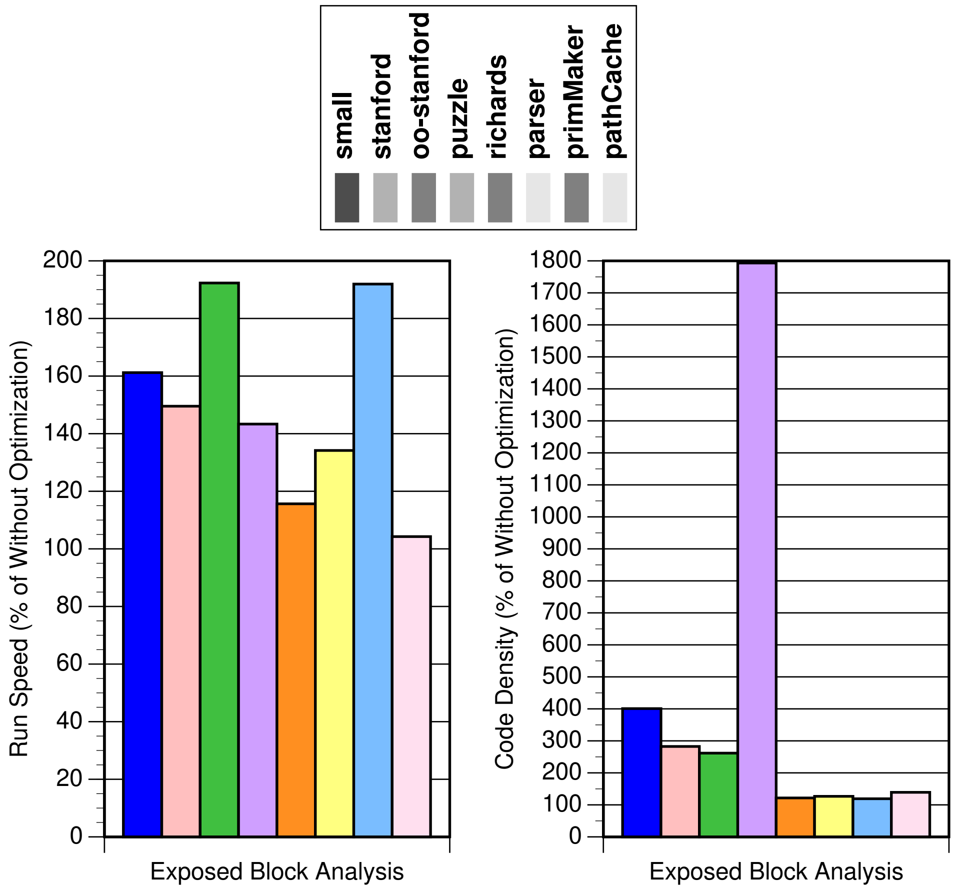


Delaying and eliminating unneeded block creations clearly boosts execution speed, frequently more than either customization or type prediction. Of course, this effect is attributable primarily to eliminating unneeded block creations entirely than simply to delaying some block creations until the latest possible moment. In any case, these measurements dramatically display the importance of optimizing block creation to get good performance in a language that relies on user-defined control structures.

Eliminating block creations saves a significant amount of compile time, speeding the compiler by more than 50% for the smaller benchmarks. Not surprisingly, eliminating the code that would have created the unnecessary blocks significantly improves compiled code space density, by more than a factor of 2.5 for the smaller benchmarks.

B.3.9.2 Exposed Block Analysis

The SELF compiler tracks which blocks have been created and passed out as run-time arguments to other methods, to reduce the number of local variables and arguments which must be considered up-level accessible. This analysis can be disabled by treating all blocks as exposed. Unfortunately, this simulation strategy does not reclaim the compile time for manipulating lists of exposed blocks, so the impact of exposed block analysis on compilation time cannot be determined. The following charts report the impact of exposed block analysis on execution speed and compiled code space efficiency.



Analyzing which blocks are exposed significantly speeds SELF programs, making some benchmarks run almost twice as fast. Exposed block analysis greatly reduces the number of local variables and arguments that must be considered visible at send points, **`_Restart`** points, and uncommon branch entry points, which in turn eliminates many unnecessary register moves. Exposed block analysis can save a lot of compiled code space, by about a factor of 3 for the smaller numerical benchmark suites. This savings comes from the extra register moves and stack accesses that exposed block analysis proves may be eliminated.

B.3.10 Common Subexpression Elimination

The SELF compiler performs common subexpression elimination, as described in sections 9.6 and 12.2. Three distinct kinds of calculations can be eliminated as redundant by the SELF compiler:

- loads and stores,
- arithmetic operations, and
- constants.

Three aspects of common subexpression elimination in the SELF compiler can improve performance. One is simply that computations are not repeated; this is the traditional benefit of common subexpression elimination. Another benefit, unique to the SELF compiler, is that any additional type information associated with the result of the original computation can be propagated to the result of the redundant computation, as described in section 9.6.2. For example, if the compiler eliminates a load instruction as redundant, then any type information known about the contents of the loaded memory cell can be propagated to the result of the eliminated load instruction. A third benefit, also unique to the SELF compiler, is that the corresponding array bounds check may be eliminated if a load or store to an array element may be eliminated. As described in section 9.6.2, this benefit exists primarily because symbolic range analysis is not implemented in the SELF compiler.

The charts on the following page display the impact of common subexpression elimination. It is easy to disable all three effects at once by simply not recording or checking for available values; results for this configuration are shown in columns labelled CSE. The individual contributions of the effects may also be measured by recording and checking for available values, but not taking advantage of a particular aspect of the information. The columns labelled CSE of Constants, CSE of Arithmetic Operations, and CSE of Memory References report the incremental effect of common subexpression elimination of only constants, arithmetic instructions, and load and store instructions, respectively. (The sum of the incremental effects of these three columns ideally should equal the incremental effect of CSE as a whole, shown in the first column.) CSE of Memory References includes the impact of three effects: eliminating memory reference instructions, propagating information about the types of the contents of memory cells, and eliminating some redundant bounds checks. The effects of these last two components are shown in columns labelled CSE of Memory Cell Type Information and CSE of Memory Cell Array Bounds Checking, respectively. (The difference between these last two columns and the CSE of Memory References column should be the incremental effect of just eliminating the memory reference instructions.) Propagated type information is ignored by introducing a new value where a memory load is replaced with an assignment node, with this value initially bound to the unknown type.

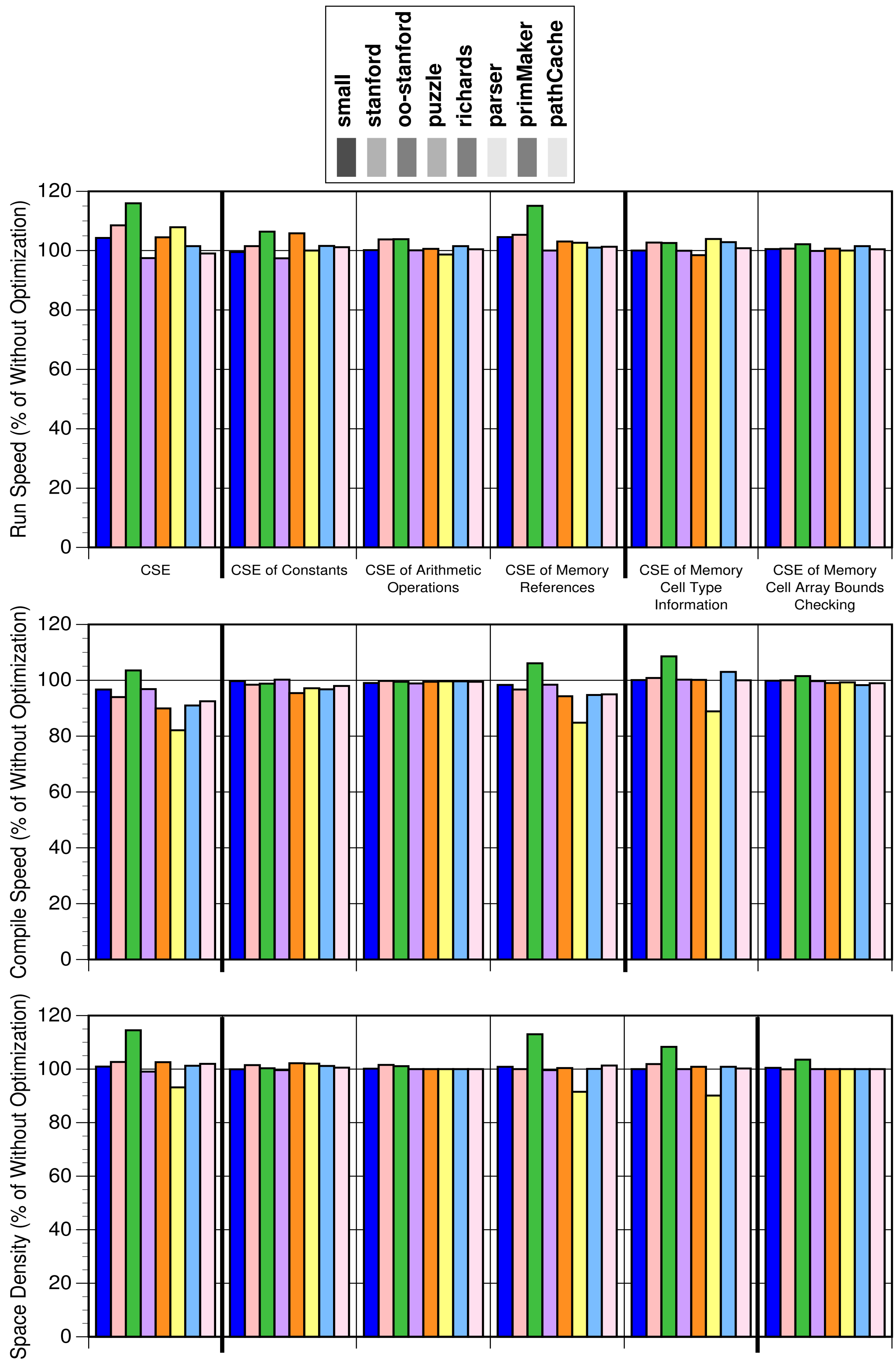
Common subexpression elimination has a relatively small effect on execution speed: less than a 15% performance improvement and occasionally a performance degradation. One cause of this poor performance is that common subexpression elimination of redundant constants is not implemented well in the current SELF system, as described in section 12.2; elimination of constants frequently replaces a two-instruction sequence (on the SPARC) that loads a 32-bit constant into a register with a one-instruction register move, instead of eliminating the loading of the constant entirely. In some cases, where the saved constant is allocated to a stack location instead of a register, performance can even slow down in the presence of common subexpression elimination of constants. Common subexpression elimination of arithmetic operations helps the Stanford integer benchmarks an average of 5%. Common subexpression elimination of memory loads and stores provides some of the best improvements, boosting the performance of the object-oriented versions of the Stanford integer benchmarks by nearly 15%.

Common subexpression elimination usually imposes a small penalty in compilation speed. The bulk of this extra cost in compilation time comes from common subexpression elimination of memory loads and stores, partially because of the extra work required to propagate the mapping from available memory cells to their contained values during type analysis and partially because of the additional inlining and other work enabled by the extra type information. Fortunately, where compilation speed drops, execution speed rises, so common subexpression elimination tends to pay for its cost in compile time with savings in execution time.

Common subexpression elimination usually saves a small amount of compiled code space, as would be expected. For the **`parser`** benchmark, however, the presence of common subexpression elimination *increases* the amount of code generated. This increase may be attributed to the extra inlining enabled by the type information available from common subexpression elimination of memory references.

B.3.11 Register Allocation

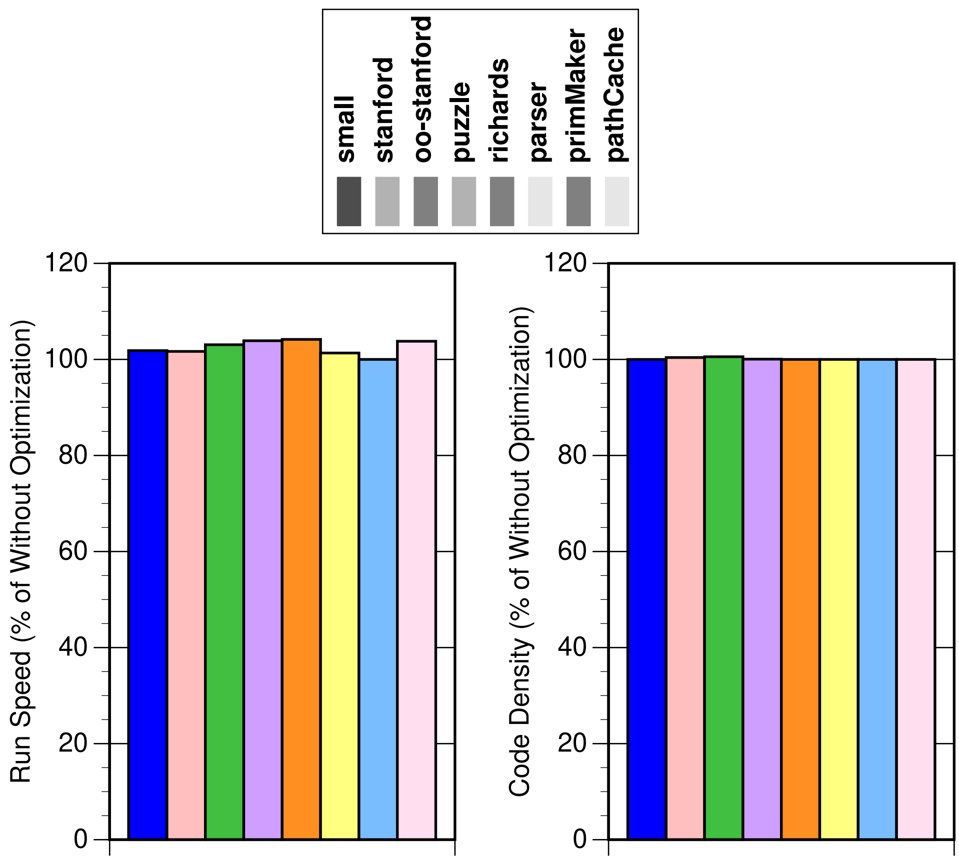
The SELF compiler maps names that are always aliases to variables, and then allocates run-time locations to these variables. This register allocation design was described in section 12.1. Unfortunately, we did not measure the effectiveness of this register allocation strategy.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Without Optimization)
0
20
40
60
80
100
120
CSE
CSE of Constants
CSE of Arithmetic Operations
CSE of Memory References
CSE of Memory Cell Type Information
CSE of Memory Cell Array Bounds Checking
Compile Speed (% of Without Optimization)
Space Density (% of Without Optimization)

B.3.12 Eliminating Unneeded Computations

The SELF compiler performs an additional pass over the control flow graph to eliminate unneeded computations, such as memory references and arithmetic operations whose results are never used, as described in section 12.3. Disabling this optimization would be easy except for two complications. Some of this pass is required for other parts of the SELF compiler to run correctly, such as eliminating unnecessary assignment nodes prior to constructing the variable lifetime conflict graph as part of register allocation. Also, other parts of the compiler expect some of this pass to be executed to get any sort of reasonable performance. For example, the initial type analysis phase expects that unnecessary loads of constants will be removed in this pass, and so the compiler feels free to insert such loads prior to message sends for variables that may later turn out to be the same constant at all sends. Since many variables are such constants, many loads are eliminated using this technique; without it some other technique would be used to eliminate these unnecessary loads.

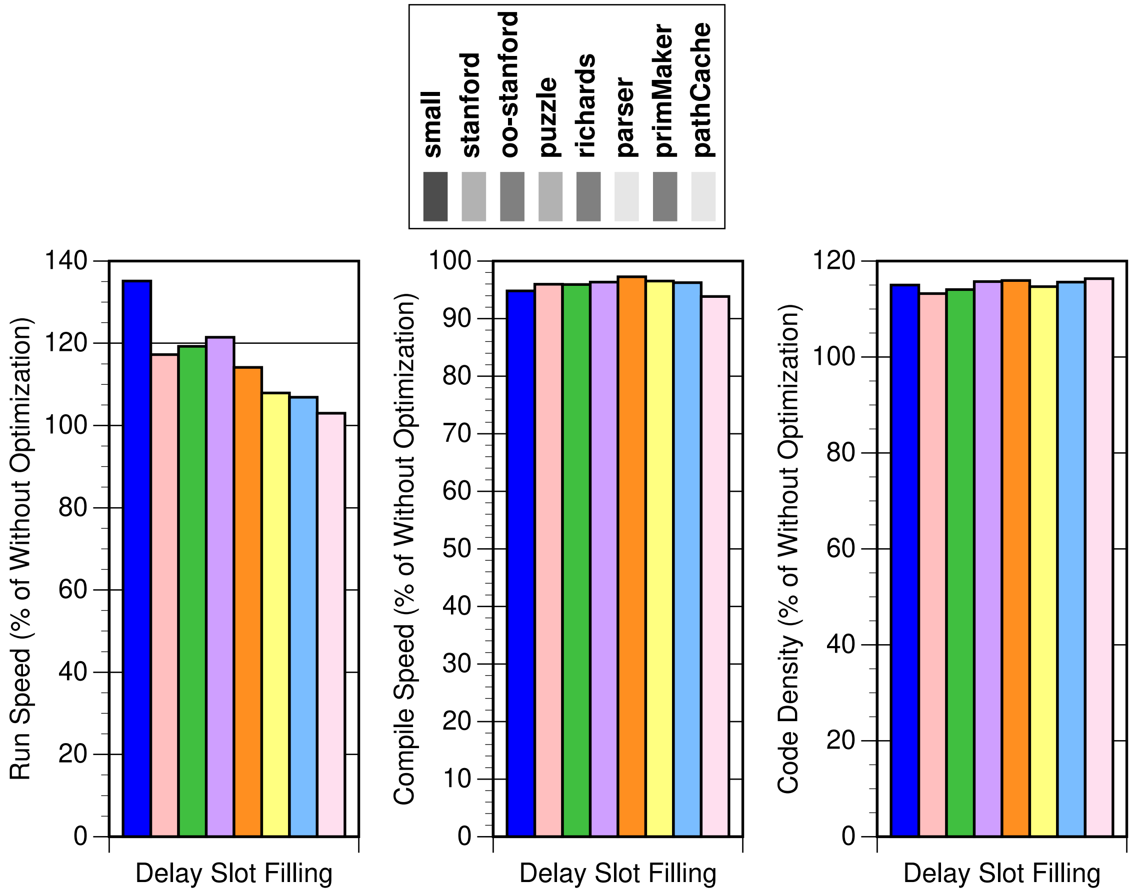
Therefore, to avoid exaggerating the importance of eliminating unneeded computations, a version of the SELF system was constructed that still eliminated assignments and loading of constants but did not eliminate unused arithmetic operations and memory loads. Compilation speed is relatively unaffected by this change, since the compiler still makes the pass to eliminate unnecessary constant loads and assignments. The following charts display the impact of eliminating unneeded arithmetic and memory loads on execution speed and compiled code space density.



Eliminating the unnecessary arithmetic and memory load instructions saves only a few percent of execution time. If eliminating unnecessary loading of constants were also disabled, the difference in execution speed would be much greater. Virtually no compiled code space is saved by this optimization, indicating that very few arithmetic or memory reference instructions were eliminated in these benchmarks.

B.3.13 Filling Delay Slots

As described in section 12.4, the SELF compiler fills delay slots on the SPARC to speed the generated code. The effect of delay slot filling on the performance of the system can be computed by constructing a version of the system than does not fill any delay slots, instead leaving a **`nop`** instruction in each delay slot. The following charts display the effect of delay slot filling.



Filling delay slots on the SPARC speeds up the benchmarks by between 5% and 35%, costs little in compile time, and reduces compiled code space requirements by almost 15%.* Given these execution speed and compiled code space benefits, delay slot filling seems to be worth its nominal compile time expense.

B.3.14 Summary of Effectiveness of the Techniques

Several techniques stand out as crucial to achieving good performance for SELF and similar languages. Inlining, deferred block creations, type prediction, and customization provide major improvements in performance. 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.

* This number can be used to place an upper bound on the average size of a basic block in the SELF system. Assuming a 15% reduction in space, one out of every 7 instructions is a delay slot that can be filled. This gives an average basic block size for SELF of at most 6 instructions (after filling delay slots).

B.4 Splitting Strategies

Splitting is one of the most important techniques used in the SELF compiler, as well as one of the most complex to implement. Several different splitting strategies were devised and implemented in the SELF compiler. In this section we explore the effectiveness of these techniques.

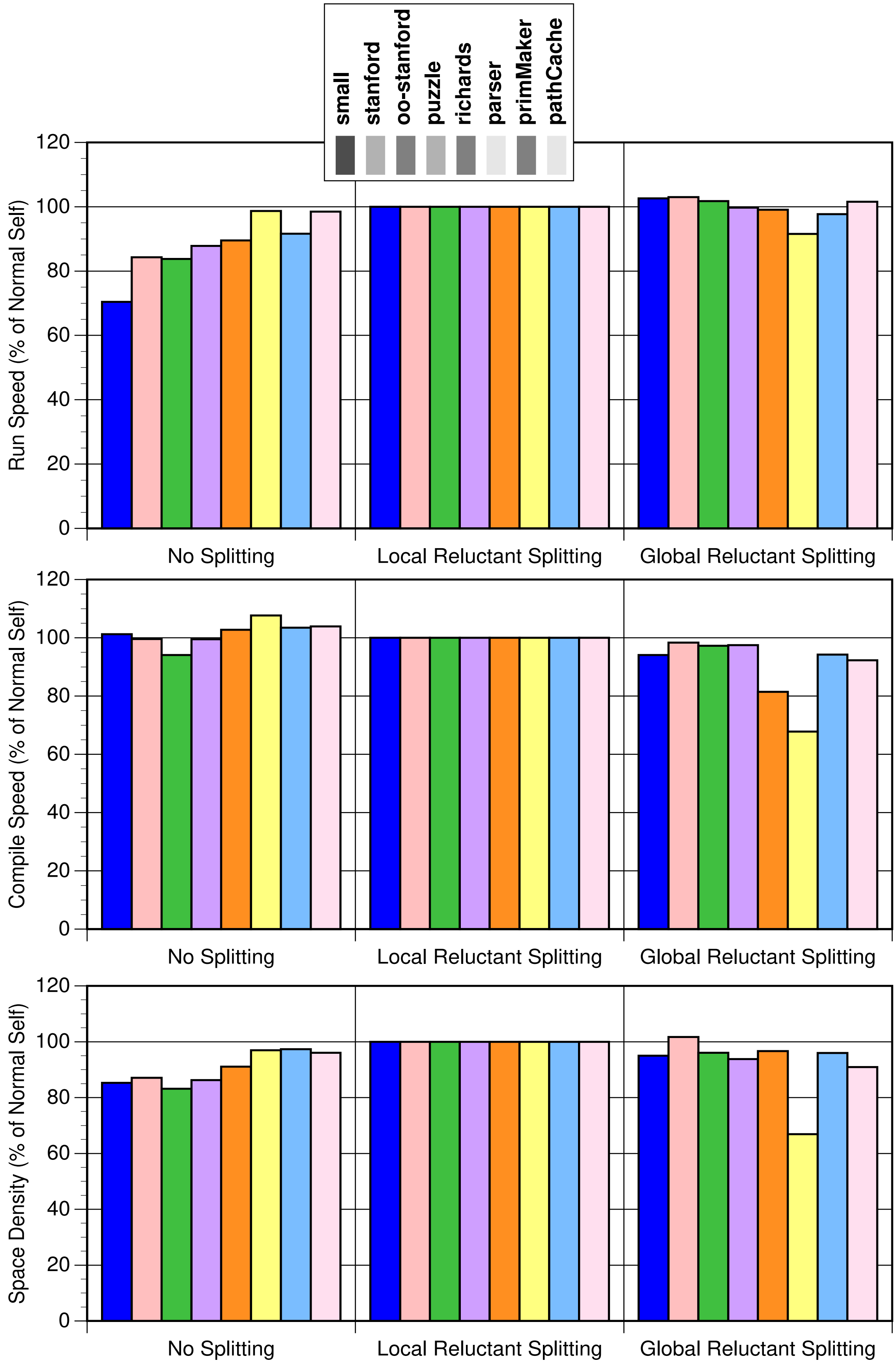
Splitting strategies can be broken down into three main approaches: no splitting, reluctant splitting (described in section 10.2), and eager splitting (described in section 10.3). Reluctant splitting itself can be divided into local reluctant splitting and global reluctant splitting, depending on how much code the compiler is willing to duplicate for a split. Eager splitting can be combined with one of three tail merging strategies: no tail merging, tail merging based on forward-computed type information, and tail merging based on reverse-computed requirements. Each of these six base splitting strategies can be combined with divided splitting (described in section 10.4) and/or lazy compilation of uncommon branches (described in section 10.5). The next few subsections explore the effectiveness of these various combinations. All measurements are reported relative to the performance of the standard SELF splitting configuration: local reluctant splitting with lazy compilation of uncommon branches but without divided splitting.

B.4.1 Reluctant Splitting Strategies

The charts on the following page illustrate the effectiveness of local reluctant splitting and global reluctant splitting in comparison to a configuration that did no splitting. These results include the effect of lazy compilation of uncommon branches. All results are relative to the performance of the standard configuration: local reluctant splitting with lazy compilation of uncommon branches.

Local reluctant splitting is clearly the best overall strategy, performing well in execution speed, compilation speed, and compiled code density. Performing no splitting at all sometimes saves some compile-time but usually incurs a significant run-time speed penalty and consumes additional compiled code space. Local reluctant splitting compares favorably to no splitting in compilation speed and code density because many of the splits performed as part of local reluctant splitting separate **`true`** results from **`false`** results after comparisons and in control structures, and performing this kind of splitting significantly *simplifies* the control flow graph (as illustrated in section 10.1), thus paying for itself through later savings in compile time and compiled code space.

Global reluctant splitting does not compensate for its larger compile time and compiled code space costs with significantly improved execution speed. In these benchmarks (and perhaps in most programs), there typically is either just a single common-case path, in which case global reluctant splitting is not needed, or a pair of common-case paths, one for a **`true`** result and one for a **`false`** result. For the **`true`** and **`false`** cases, the result of the comparison or boolean function that produced the two paths is usually consumed by the immediately following message, such as an **`ifTrue:`** message, and so local reluctant splitting is adequate to handle this kind of situation. Thus, for these benchmarks, global reluctant splitting provides unneeded extra power. As we will argue in the next subsection, global reluctant splitting would be more important if lazy compilation of uncommon branches had not been devised; lazy compilation splits off the uncommon-case paths that would have been split off, albeit less effectively, by global reluctant splitting. Even though global reluctant splitting does not seem useful in today's system with local reluctant splitting and lazy compilation, in the future multiple common-case paths may be more frequent as other implementation techniques are incorporated [HCU91], and global reluctant splitting may then begin to show significant improvement over local reluctant splitting.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
0
20
40
60
80
100
120
No Splitting
Local Reluctant Splitting
Global Reluctant Splitting

B.4.2 Lazy Compilation of Uncommon Branches

The charts on the following page report the performance of various reluctant splitting strategies both with and without lazy compilation of uncommon branches. All results are relative to the performance of the standard configuration: local reluctant splitting with lazy compilation of uncommon branches.

Lazy compilation of uncommon cases improves execution performance for almost all splitting configurations and benchmarks, as can be seen by comparing the results for each ... (Not Lazy) column to the results for the corresponding ... (Lazy) column. This speed-up may be attributed to two major factors: providing the effect of divided splitting and simplifying the register allocation problem. The largest effects occur for no splitting and local reluctant splitting; these two strategies benefit the most from the additional divided splitting effect. All strategies benefit from better register allocation enabled by lazy compilation.

The largest impact on execution speed occurs for the more numerically-oriented benchmarks. This is as expected, since in these benchmarks many messages get inlined down to primitives that introduce primitive failure uncommon branches (such as **+** messages getting inlined down to **`_IntAdd:`** primitive calls, each of which may fail with an overflow error), and many type prediction type tests get inserted (such as integer type tests before **+** messages) that each create an uncommon branch. The larger number of uncommon branch entries in these benchmarks provides more opportunities for lazy compilation.

Lazy compilation makes a dramatic improvement in both compilation speed and compiled code space efficiency, across the board for all base splitting strategies and benchmarks. As expected, lazy compilation improves performance most for the smaller, more numerical benchmarks.

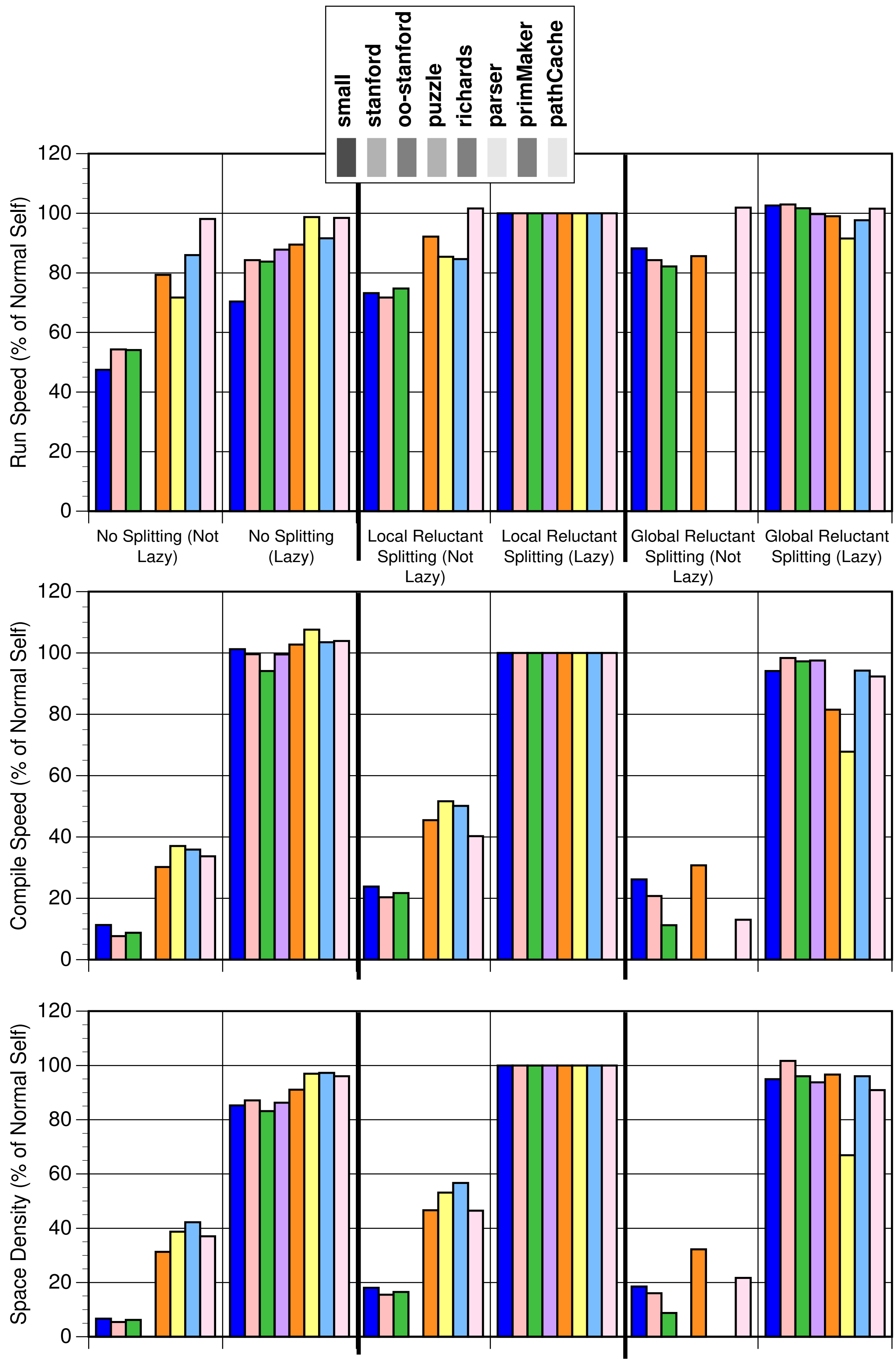
Lazy compilation of uncommon cases obviates much of the need for more sophisticated base splitting techniques such as global reluctant splitting. If neither lazy compilation nor divided splitting were implemented, then global reluctant splitting would offer performance improvements over local reluctant splitting, as can be seen by comparing the Local Reluctant Splitting (Not Lazy) results to the Global Reluctant Splitting (Not Lazy) results. However, since in the present system only one common case path normally is generated, global reluctant splitting does not provide much additional functionality.

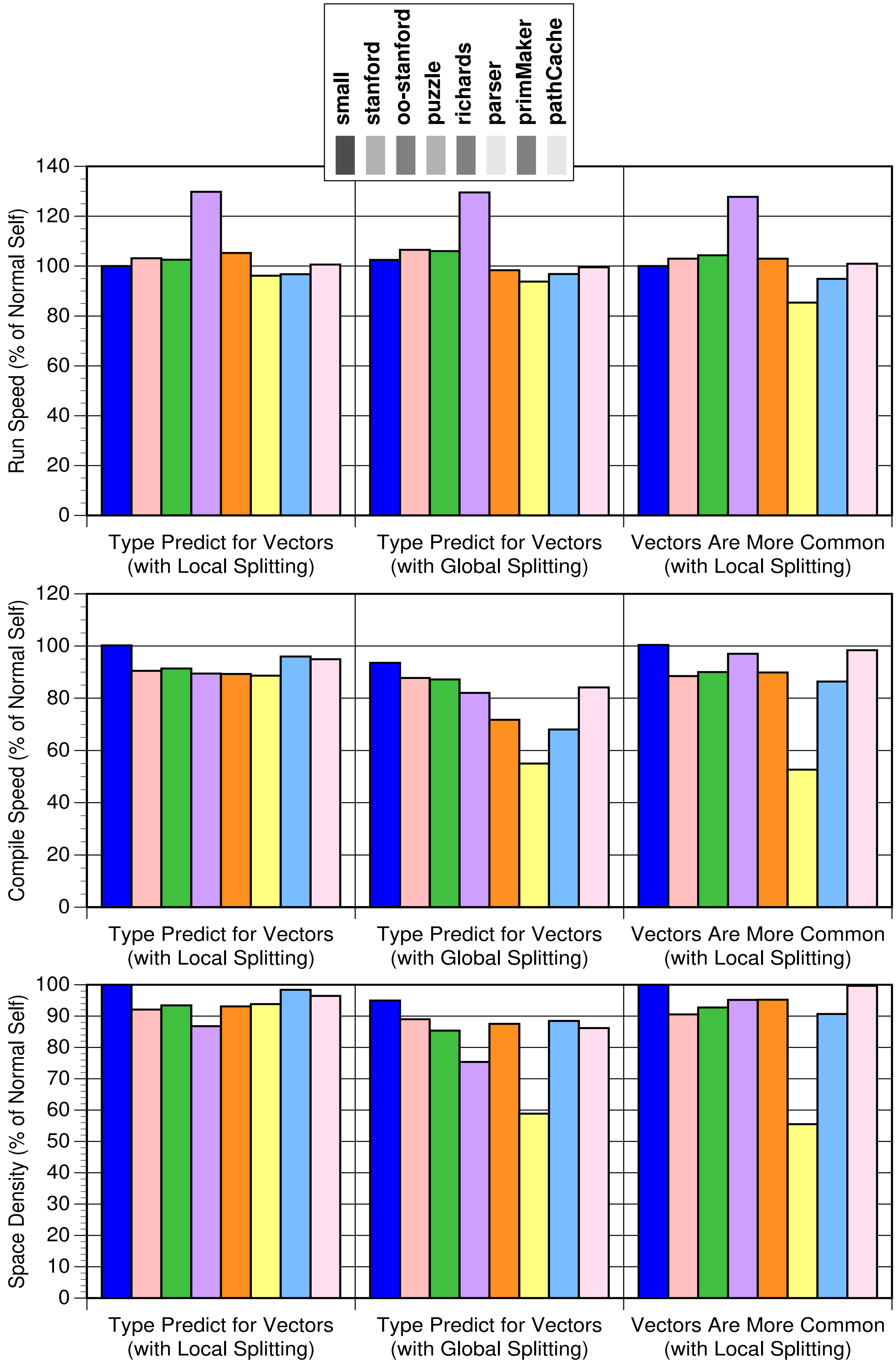
B.4.3 Vector Type Prediction

When configuring the SELF system, one choice that must be made (today at least) is the set of message names and corresponding receiver types that are to be type-predicted. For most messages, the choice is fairly obvious: predict **`true`** or **`false`** for the receiver of **`ifTrue:`** and predict integer for the receiver of **+**, for instance. Unfortunately, some choices are not so clear-cut. For example, what should the compiler predict for the **=** message? Currently, the compiler predicts that the receiver often is an integer (and consequently inserts a type-test for an integer receiver), but since many kinds of objects other than integers are compared for equality, the non-integer branch is not treated as uncommon but instead gives rise to a second common-case path.

The compiler faces a similar problem for messages like **`at:`** and **`at:Put:`**. For example, the compiler could predict that the receiver of these messages is likely to be the built-in fixed-length vector type. All the arrays manipulated by the micro-benchmarks and the Stanford integer benchmarks are of this type, so this prediction should improve the performance of these benchmarks. However, in other parts of the system, the receivers of **`at:`** and **`at:Put:`** are more likely to be dictionaries or other kinds of keyed collections, not simple fixed-length vectors. The current standard SELF configuration does no type prediction for **`at:`** and **`at:Put:`**, in consideration of the more object-oriented programs currently being run in the SELF system, but other kinds of programs (including the benchmarks) might benefit from type predicting fixed-length vector receivers for **`at:`** and **`at:Put:`**.

To determine the improvement that could be gained from vector type prediction, we measured the performance of three additional configurations. We extended the standard configuration based on local reluctant splitting to additionally type-predict for vectors; one version treats non-vector cases as uncommon, while another treats non-vector cases as giving rise to a second common-case path. Since global reluctant splitting should shine in the presence of multiple common-case paths, we also measured a configuration based on global reluctant splitting which type-predicts for vectors and treats non-vector cases as additional common-case paths. The charts on page 195 report the performance of these three vector type prediction configurations relative to that of the standard configuration based on local reluctant splitting and no vector type prediction.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
0
20
40
60
80
100
120
No Splitting (Not Lazy)
No Splitting (Lazy)
Local Reluctant Splitting (Not Lazy)
Local Reluctant Splitting (Lazy)
Global Reluctant Splitting (Not Lazy)
Global Reluctant Splitting (Lazy)


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
Type Predict for Vectors (with Local Splitting)
Type Predict for Vectors (with Global Splitting)
Vectors Are More Common (with Local Splitting)

Type prediction for vectors improves performance for some of the benchmarks. Most individual benchmarks are unchanged, but a few benchmarks such as **`intmm`**, **`oo-intmm`**, and **`puzzle`** improve by between 20% and 30% (raw data may be found in Appendix C). Treating non-vector cases as uncommon only makes a difference on the **`oo-intmm`** benchmark, boosting its performance another 10%. Global reluctant splitting is just as effective as treating non-vectors as uncommon, verifying that global reluctant splitting can provide improved performance in the presence of multiple common-case paths.

Unfortunately, type predicting for vectors slows down the larger, more object-oriented benchmarks in which the predictions are always wrong. When non-vector receivers are considered uncommon, mispredicting forces the compiler to generate less optimized uncommon branch extensions, further slowing down execution performance; the **`parser`** benchmark runs 15% slower in this mode than in the normal configuration. Since we expect most SELF programs to be more like **`parser`** than **`puzzle`**, the normal SELF system does not type-predict for vectors. However, in environments or applications where vector receivers were more common, vector type prediction could significantly improve performance. Developing a system in which some parts type-predict for vectors while others do not is an active area of current research [HCU91].

Type predicting for vectors uniformly slows down compilation. Usually this slow-down is less than 20%, but when mispredictions force the compiler to generate uncommon branch extension methods (when non-vector cases are considered uncommon), compile times can be twice as long as with the normal configuration. In fact, these potentially lengthy compile pauses are the primary reason we currently do not type-predict vector receivers.

Not surprisingly, vector type prediction takes up more compiled code space, but usually not more than 10% over the non-predicting configuration. When mispredictions and lazy compilation cause uncommon branch extensions to be created, however, code space consumption can double.

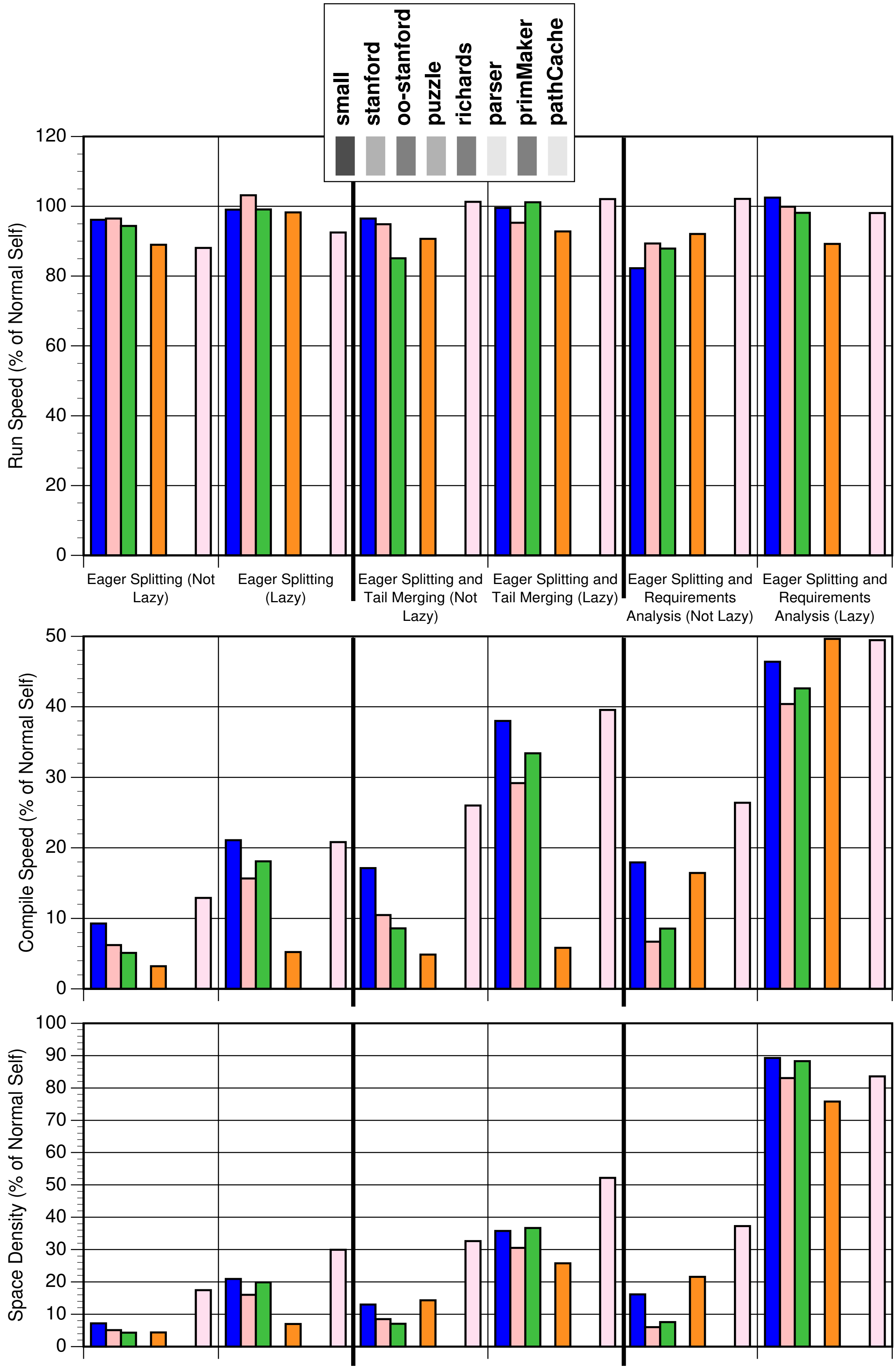
B.4.4 Eager Splitting Strategies

The charts on the following page report execution performance for the various eager splitting configurations relative to that of the standard configuration, local reluctant splitting with lazy compilation of uncommon branches.

The various tail merging strategies are designed to save compile time and compiled code space while producing the same execution speed. These expectations are borne out by the results. Different tail merging strategies achieve roughly the same execution speeds, but requirements-analysis-based tail merging, the most sophisticated tail merging technique, achieves much better compilation speeds and compiled code densities than the simpler techniques; the extra reverse pass over the control flow graph that is part of requirements analysis easily pays for itself in reduced overall compile time.

Lazy compilation slightly improves the execution performance of eager splitting. Since eager splitting already incorporates divided splitting, this potential benefit of lazy compilation is unneeded. Compilation speed improves more dramatically with lazy compilation of uncommon cases. This speed-up becomes more pronounced as the tail merging strategy becomes more sophisticated. Lazy compilation makes less of an impact for the simpler tail merging strategies since those strategies spend more of their compile time compiling common-case parts of the control flow graph, and so there is less compile time spent on uncommon branches to be eliminated. Similarly, lazy compilation dramatically improves compiled code space efficiency, especially for eager splitting with requirements-based tail merging.

Unfortunately, overall eager splitting provides no execution speed benefits compared to reluctant splitting and at best more than doubles the compilation time costs; in some cases the benchmarks could not even be compiled using eager splitting because they would have needed too much compiler temporary data space. Clearly, eager splitting is not practical as currently implemented.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
0
20
40
60
80
100
120
Eager Splitting (Not Lazy)
Eager Splitting (Lazy)
Eager Splitting and Tail Merging (Not Lazy)
Eager Splitting and Tail Merging (Lazy)
Eager Splitting and Requirements Analysis (Not Lazy)
Eager Splitting and Requirements Analysis (Lazy)
Compile Speed (% of Normal Self)
0
10
20
30
40
50
Space Density (% of Normal Self)
0
10
20
30
40
50
60
70
80
90
100

B.4.5 Divided Splitting

The charts on the following page report the performance of various reluctant splitting strategies both with and without divided splitting. All results are relative to the performance of the standard configuration: local reluctant splitting with lazy compilation of uncommon branches and without divided splitting.

Divided splitting makes no difference when used in conjunction with lazy compilation of uncommon branches, as can be seen by comparing the results for columns labelled ... (Lazy) to the results for columns labelled Divided ... (Lazy). This is as expected, since lazy compilation achieves the same beneficial effect on type analysis as does divided splitting but in a more efficient manner. For the strategies without lazy compilation, divided splitting speeds execution significantly, as can be seen by comparing the results for columns labelled ... (Not Lazy) to the results for columns labelled Divided ... (Not Lazy). The advantages provided by divided splitting decrease with the sophistication of the base splitting technique. Divided splitting usually slows compilation and consumes more compiled code space, as expected.

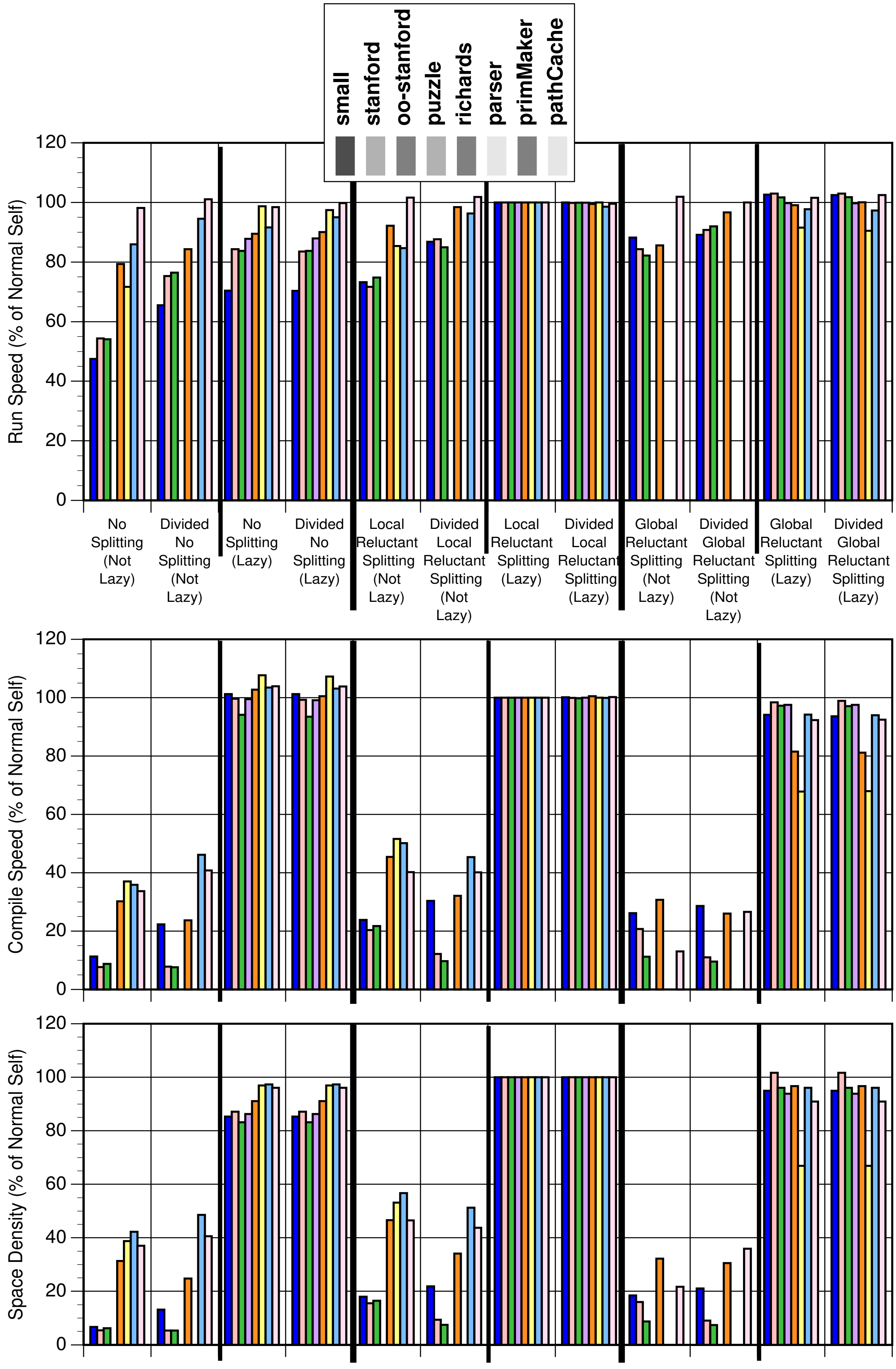
Based on these results, divided splitting provided no additional benefits over lazy compilation of uncommon cases. However, in environments where lazy compilation of uncommon cases was impossible, divided splitting can offer improved performance.

B.4.6 Summary of Splitting Strategies

Local reluctant splitting combined with lazy compilation of uncommon cases is the most effective splitting strategy, optimizing all of execution speed, compilation speed, and compiled code space efficiency. Lazy compilation provides dramatic, near order-of-magnitude improvements in compilation speed and code density, and even boosts execution speed and obviates the need for divided splitting. Global reluctant splitting provides slight improvements in performance for these benchmarks but at a noticeable decrease in compilation speed. Eager splitting as presently implemented provides no significant performance improvements and sacrifices at least half of the compilation speed of local reluctant splitting. Vector type prediction can improve performance for some benchmarks but slows down other benchmarks.

In the chart on page 200, we summarize the effectiveness of various splitting strategies and their trade-offs between execution speed and compilation speed by plotting the execution speed and compilation speed of several splitting strategies on a two-dimensional scatter chart; relative compiled code space efficiency is very nearly directly proportional to relative compilation speed so a third dimension is not necessary. To make the chart more readable, only selected configurations are included.

The cluster of strategies in the upper-right corner of the chart (in the "good" region) all use reluctant splitting and lazy compilation of uncommon branches. The other strategies produce no better execution speeds with poor compilation speeds.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
0
20
40
60
80
100
120
No Splitting (Not Lazy)
Divided No Splitting (Not Lazy)
No Splitting (Lazy)
Divided No Splitting (Lazy)
Local Reluctant Splitting (Not Lazy)
Divided Local Reluctant Splitting (Not Lazy)
Local Reluctant Splitting (Lazy)
Divided Local Reluctant Splitting (Lazy)
Global Reluctant Splitting (Not Lazy)
Divided Global Reluctant Splitting (Not Lazy)
Global Reluctant Splitting (Lazy)
Divided Global Reluctant Splitting (Lazy)

paste chart-page-200.ps here

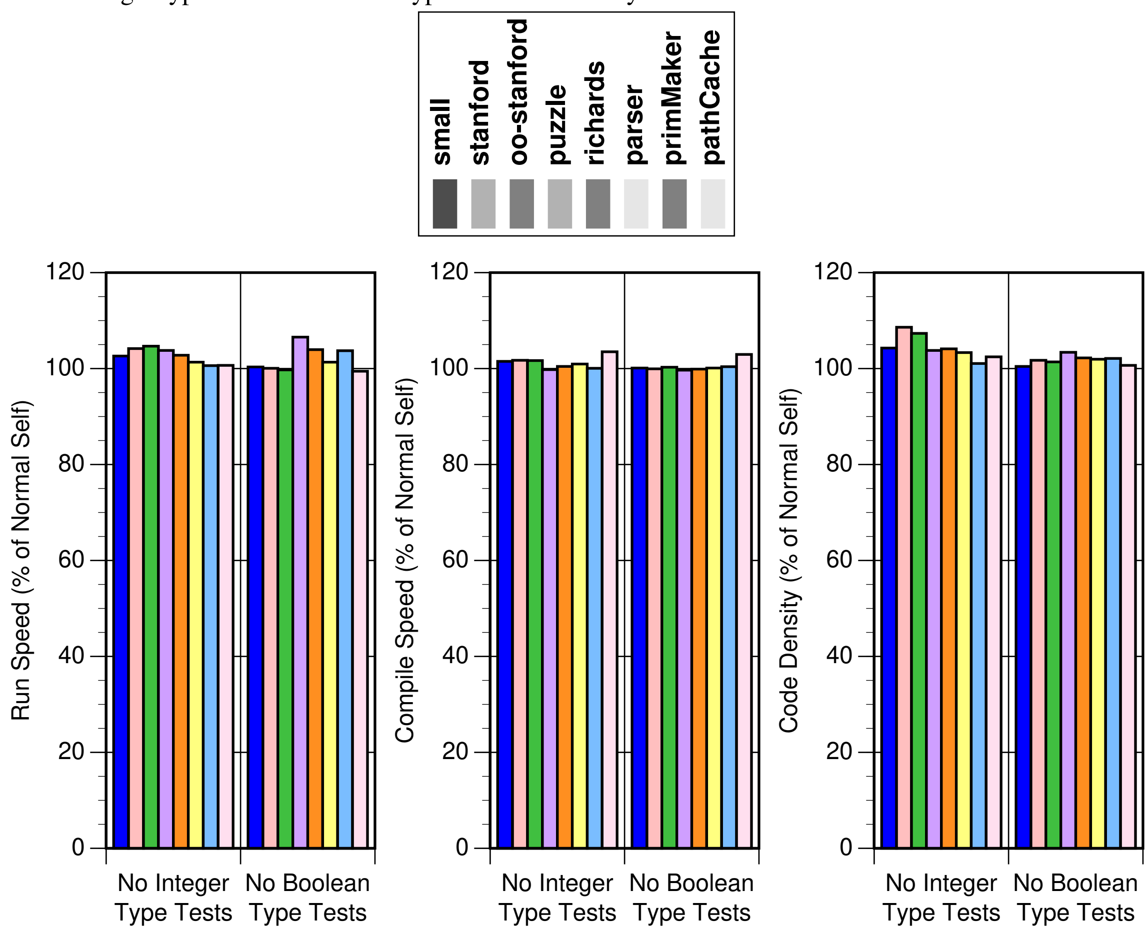
B.5 Some Remaining Sources of Overhead

This section explores in detail each of the remaining sources of overhead that we were able to measure in the SELF system that are not present in a traditional optimized language implementation. These results were summarized in section 14.4.

B.5.1 Type Tests

The SELF compiler inserts extra run-time type tests that are not present in the output of the optimizing C compiler. These tests are inserted as part of type prediction for messages like **`+`** and **`ifTrue:`** and as part of type-checking of arguments to primitives. Language features like message passing, dynamic typing, generic arithmetic, and safe primitives, present in SELF but not in C, incur this extra overhead.

We can measure the cost of these run-time type tests by constructing a version of SELF that does not generate any type checks but instead assumes that the type tests would always succeed. This configuration obviously is unsafe, but none of the benchmarks happen to fail any type tests and so do not break under this configuration. The charts below report the costs in execution speed, compilation speed, and compiled code space for run-time type tests and checks; we report the costs of integer type tests and boolean type tests individually.

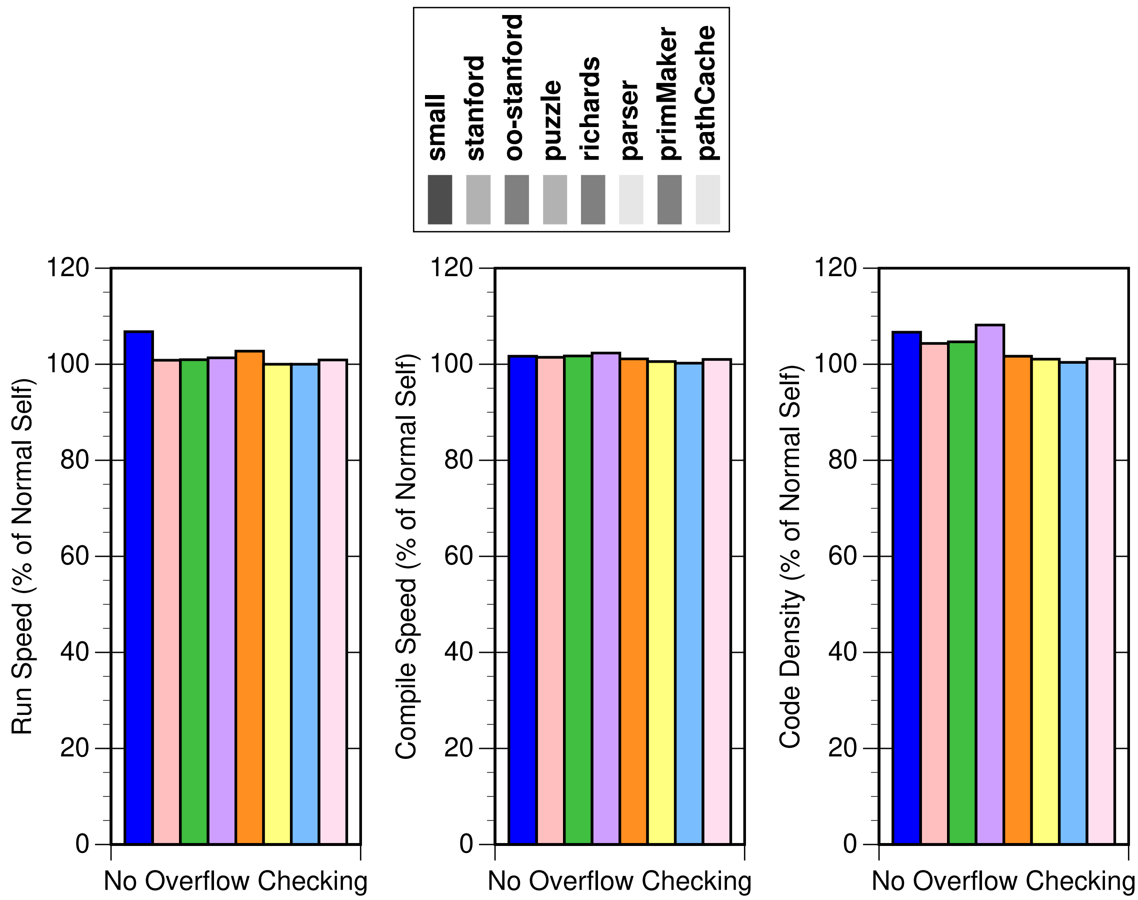


Type tests increase execution time by no more than 10% for these benchmarks, take up little extra compile time, and add no more than 10% additional compiled code space. Type analysis, splitting, and lazy compilation of uncommon cases are largely responsible for the relatively low number of type tests that remain in compiled SELF code.

B.5.2 Overflow Checking

The SELF compiler sometimes generates an overflow check after an integer arithmetic instruction to check for primitive failure of the corresponding integer arithmetic primitives. By handling this primitive failure, SELF programs can (and do) support generic arithmetic. Generic arithmetic traditionally has been an expensive language feature; section 14.2 showed that the performance of T programs that use generic arithmetic can be half that of T programs that avoid generic arithmetic. Much of the added cost of generic arithmetic involves the extra type tests associated with checking for integer arguments to generic arithmetic operations; this overhead was measured in the previous section. The remaining cost of generic arithmetic is incurred by the extra overflow checks which are not generated by the integer-specific arithmetic supported by traditional languages.

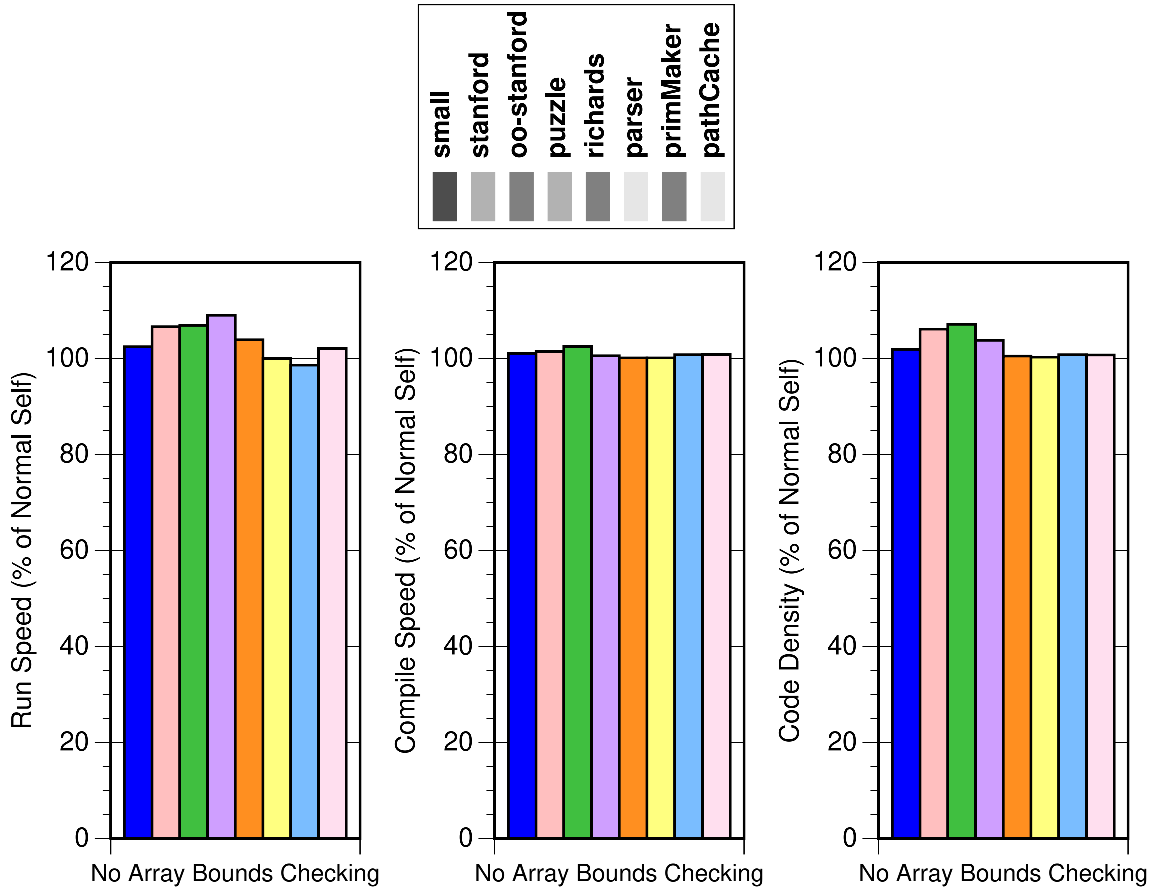
We can measure the cost of overflow checking in SELF simply by not checking for overflows. Again, this configuration is unsafe, but none of the benchmarks overflow any arithmetic operations. The charts below report the cost of overflow checking.



Overflow checking slows execution by less than 5% for all but the smallest benchmarks. However, since the register allocator frequently leaves an extra register move behind even when overflow checks are removed, the cost of overflow checking may be understated by these results by up to a factor of two. Overflow checking imposes negligible compile-time cost and a small space cost.

B.5.3 Array Bounds Checking

SELF array accessing primitives always verify that accesses lie within the bounds of the array. We can measure the cost of array bounds checking in SELF by not checking for access out-of-bounds in the generated code. The following charts display the cost of array bounds checking.



Array bounds checking imposes a modest run-time performance cost, between 5% and 10% overhead for those benchmarks manipulating arrays. We think that much of this overhead could be eliminated by applying more sophisticated integer subrange analysis using symbolic bounds, but the complexity of this technique may not be worth the apparently modest improvement in execution time. Very little compile time is used for generating code to check for out-of-bounds array accesses. Some compiled code space is required to support array bounds checking in the SELF implementation, but this space overhead is less than 10%. The cost would be much higher if the compiler did not compile uncommon cases lazily.

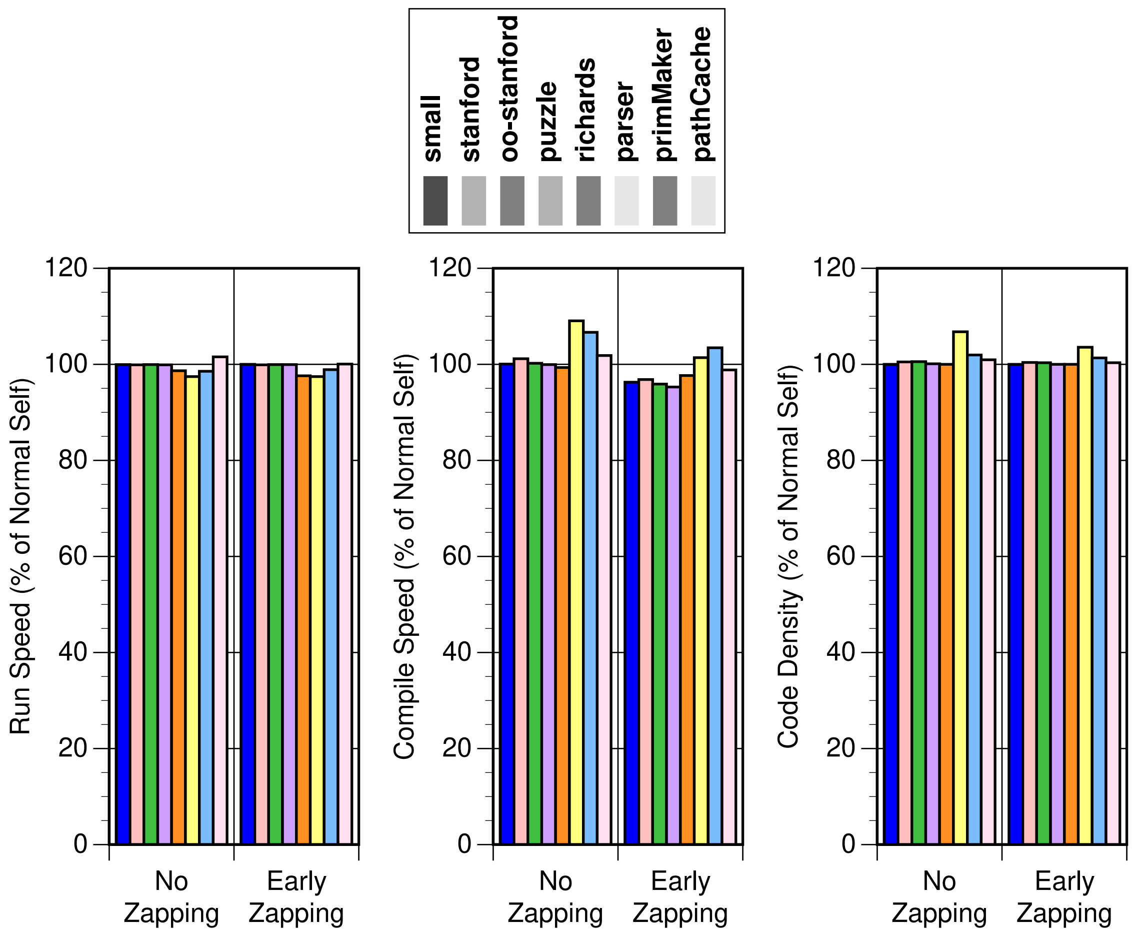
B.5.4 Block Zapping

The current SELF implementation does not allow a block to be invoked after its lexically-enclosing activation record has returned. To enforce this restriction, the compiler generates extra code that "zaps" blocks once they become uninvokable, as described in section 6.3.2. This zapping cost could become fairly expensive. Block zapping involves both extra run-time code needed to zero the frame pointer of any block that might have been created and extra registers or stack locations to hold the blocks until they are used by the zap code. Since block `value` methods themselves do not require explicit run-time code to test for a zero frame pointer, instead relying on the machine's addressing and protection hardware to trap references to illegal addresses, this zapping architecture imposes no run-time overhead on block invocation.

To gauge the cost of this design, we measured three configurations of the SELF system, each with a different rule for zapping blocks:

- no zapping,
- early zapping (block lifetimes extend to the end of the message in which they are initially an argument), and
- late zapping (block lifetimes extend to the end of the scope in which they are created); this is the standard configuration.

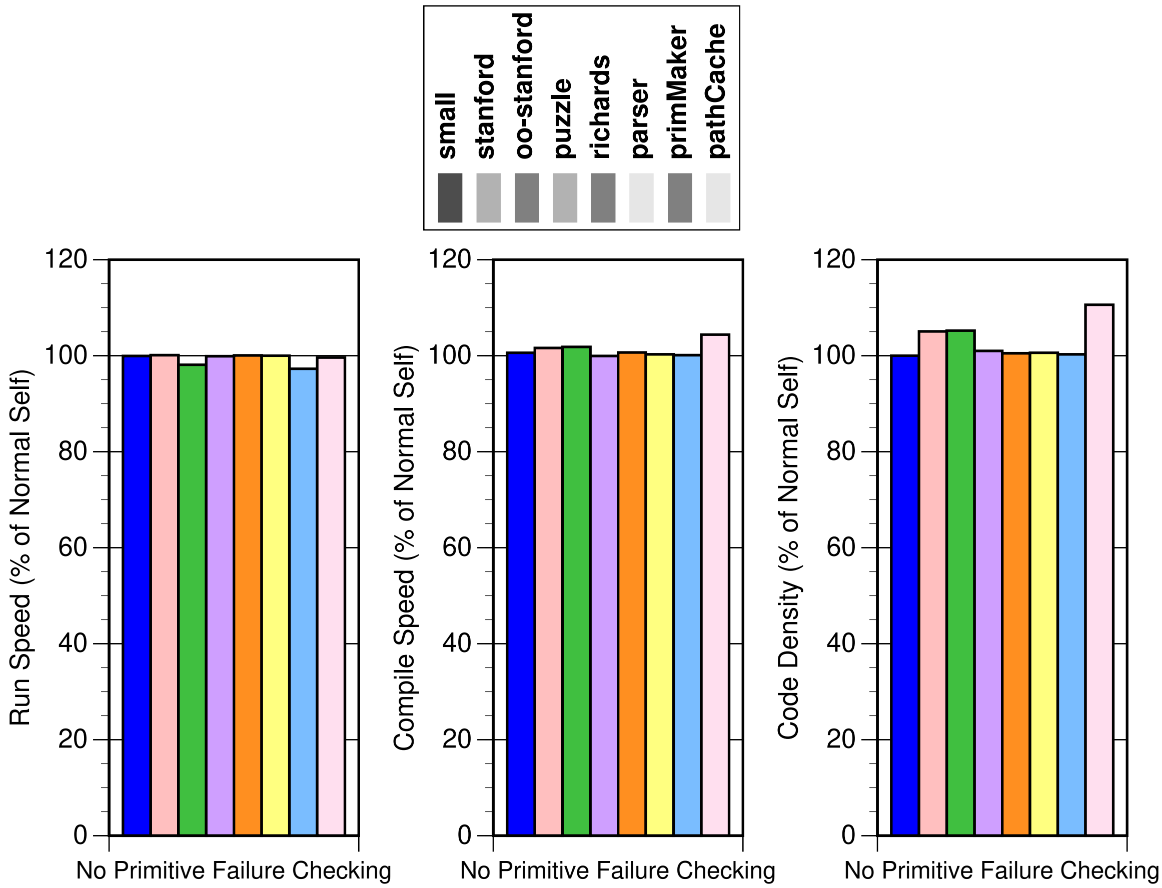
Early zapping was proposed as an alternative to late zapping that might be less expensive, particularly in terms of register usage. The following charts report the performance of no zapping and early zapping relative to late zapping, the standard configuration.



Surprisingly, block zapping has a negligible impact on execution speed of these benchmarks. The slight slow-downs for some of the benchmarks may be caused by unlucky interactions with other parts of the system; performance should only improve with these alternative zapping implementations. As expected, the SELF compiler is faster when it does not bother generating any zap code. Early zapping slows down the compiler relative to late zapping; the data structures and algorithms used by the compiler to generate the early zapping code are more complex than those for late zapping. Both no zapping and early zapping have slightly better compiled code space efficiencies than late zapping, but by a few percent at most. Based on these results, the safety and greater flexibility of late zapping make it the preferable block zapping strategy. Some future implementation of SELF might support true non-LIFO blocks (fully upward closures) and dispense entirely with the need for block zapping, simplifying this part of the compiler's implementation in the process.

B.5.5 Primitive Failure Checking

If the SELF compiler implements a primitive by calling a routine in the virtual machine (rather than generating special inlined code for the primitive), the compiler generates code after the primitive call that checks for the primitive's failure by testing for a special return value. This run-time overhead could be avoided by an alternate calling convention for such primitives that used different return offsets for successful returns and failing returns. To determine whether this change would be useful, we need to know the cost of the current design. We can compute this cost simply by not checking for primitive failures, assuming that all such primitives succeed. The following charts report the cost of external primitive failure checking as currently implemented.

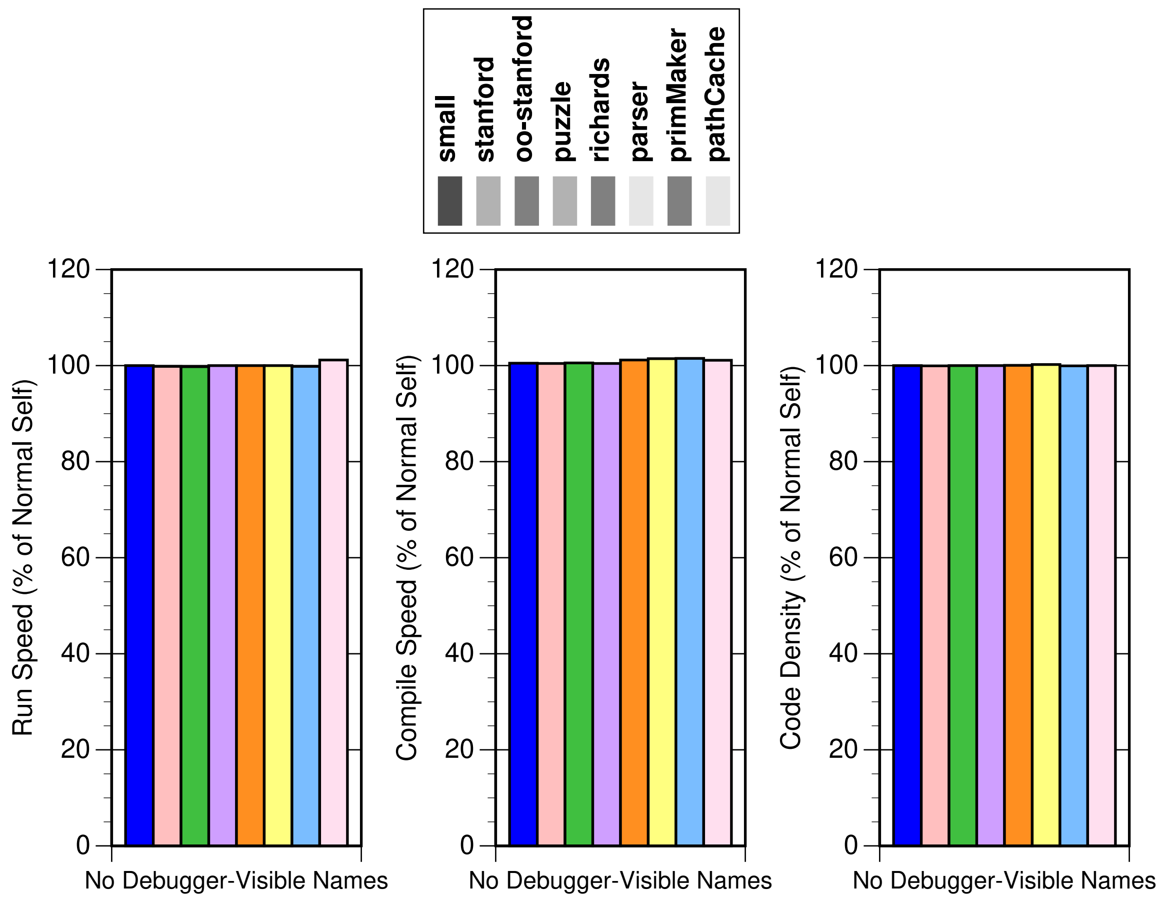


Primitive failure checking for non-inlined primitives has no effect on execution performance for these benchmarks. The slight slow-down for the **`oo-stanford`** and **`primMaker`** benchmarks may be unfortunate interactions with the register allocator, since removing the run-time check should only improve performance. Primitive failure checking imposes a slight cost in compile time and increases space costs by up to 10%. These costs do not seem significant enough to justify optimizing the return sequence of external primitives.

B.5.6 Debugger-Visible Names

In section 5.1 we argued that a language implementation should support debugging of the entire program at the level of the source code, with all optimizations and other implementation artifacts hidden from the programmer. This requirement restricts the kinds of optimizations that can be performed, thus possibly degrading performance over a system that did not support source-level debugging.

Some of these costs cannot be measured easily in our system, such as the cost of not performing tail call elimination or of not reordering code. Fortunately, at least one of the costs attributable to the need for debugging can be measured. In the SELF system, the programmer can get a complete view of the state of a suspended process, including the values of all data slots in activation records (i.e., the contents of local variables and arguments of source-level stack frames), as described in section 13.1. This requires that the compiler ensure that the contents of a variable is always up-to-date and available as long as the debugger might access it, even if the compiled code has no more need of the variable. This imposes a cost in terms of registers that are used and cannot be reallocated to other expressions, possibly causing some expressions to be spilled out to the stack. We measured the cost of this variable lifetime extension by configuring the compiler to ignore the effect of the debugger when computing the lifetime of variables, freeing up registers as soon as the compiled code has no more use of the variable, and report the effects in the charts below.

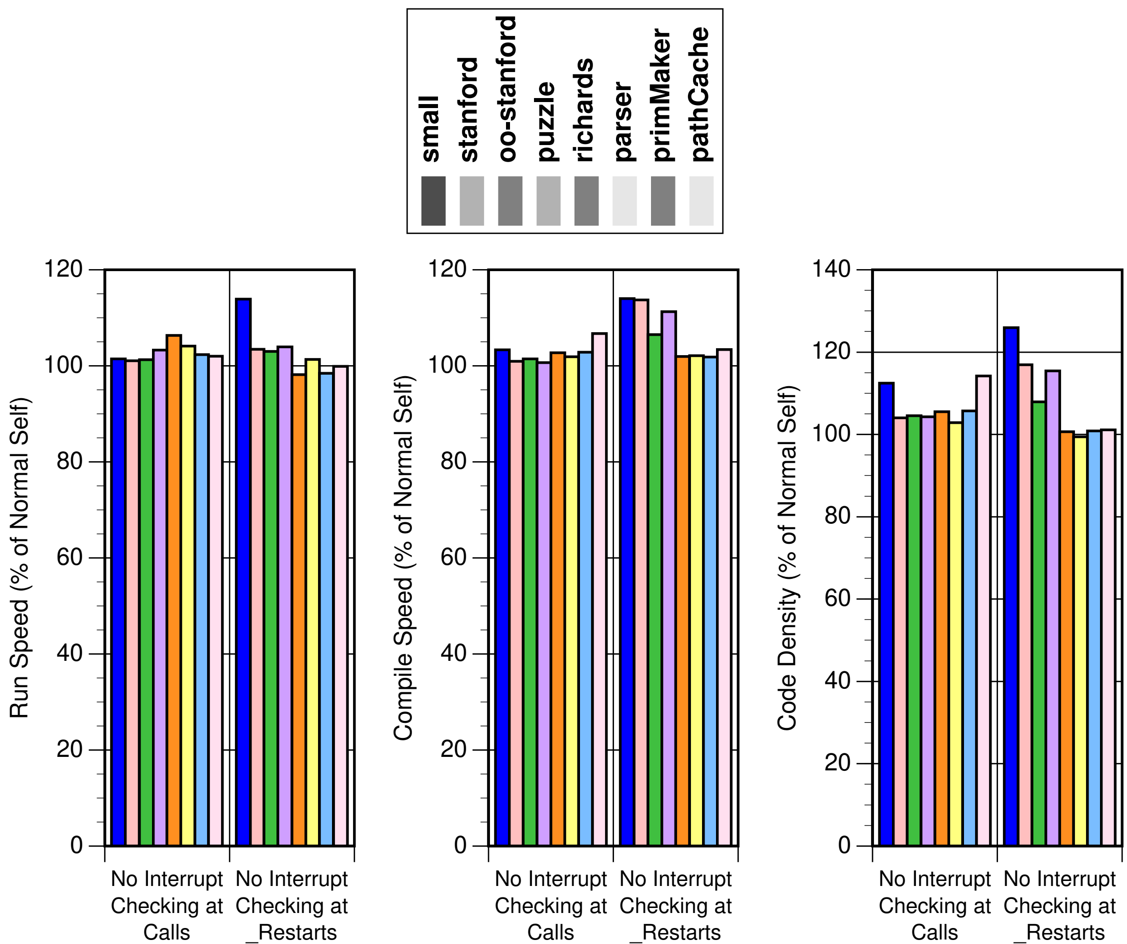


Supporting the debugger's ability to view all source-level visible names whether or not the executing code requires a name imposes no execution-time cost for these benchmarks. This is partially attributable to the presence of hardware register windows on the SPARC (since leaving an unused variable in a register is free while saving and restoring an unused variable across calls is not), partially attributable to the presence of interruption points and uncommon branch entry points that force many names to be maintained anyway, and partially attributable to the infrequency of variable names becoming unused well prior to the end of their scope, perhaps because methods in SELF are typically much shorter than procedures in a traditional language. Ensuring that names stay live throughout their visible lifetimes takes a negligible amount of extra compile time and a negligible amount of compiled code space. Clearly, this aspect of the programming environment can be supported at no cost.

B.5.7 Interrupt Checking and Stack Overflow Checking

The SELF run-time system handles interrupts via polling, with the compiler generating code at method entry and **`_Restart`** points (loop tails) to check for interrupts, as described in section 6.3. The cost associated with these checks might be avoided by an alternative interrupt architecture that does no polling but instead backpatches instructions ahead of the current program counter to call the interrupt handler at exactly the same point that the polling code would have detected the interrupt. Also, since the interrupt check at each method entry doubles as a check for stack overflow, this use of polling would need to be eliminated by read-protecting the page after the maximum stack size and using the page protection hardware to detect stack overflow.

Since the costs current associated with interrupt polling could probably be avoided by a more sophisticated run-time system design, it is important to determine how costly is the polling overhead in the current SELF implementation. We can measure the cost of polling simply by not generating any polling code and ignoring interrupts. The following charts report the costs for the two sources of polling overhead.



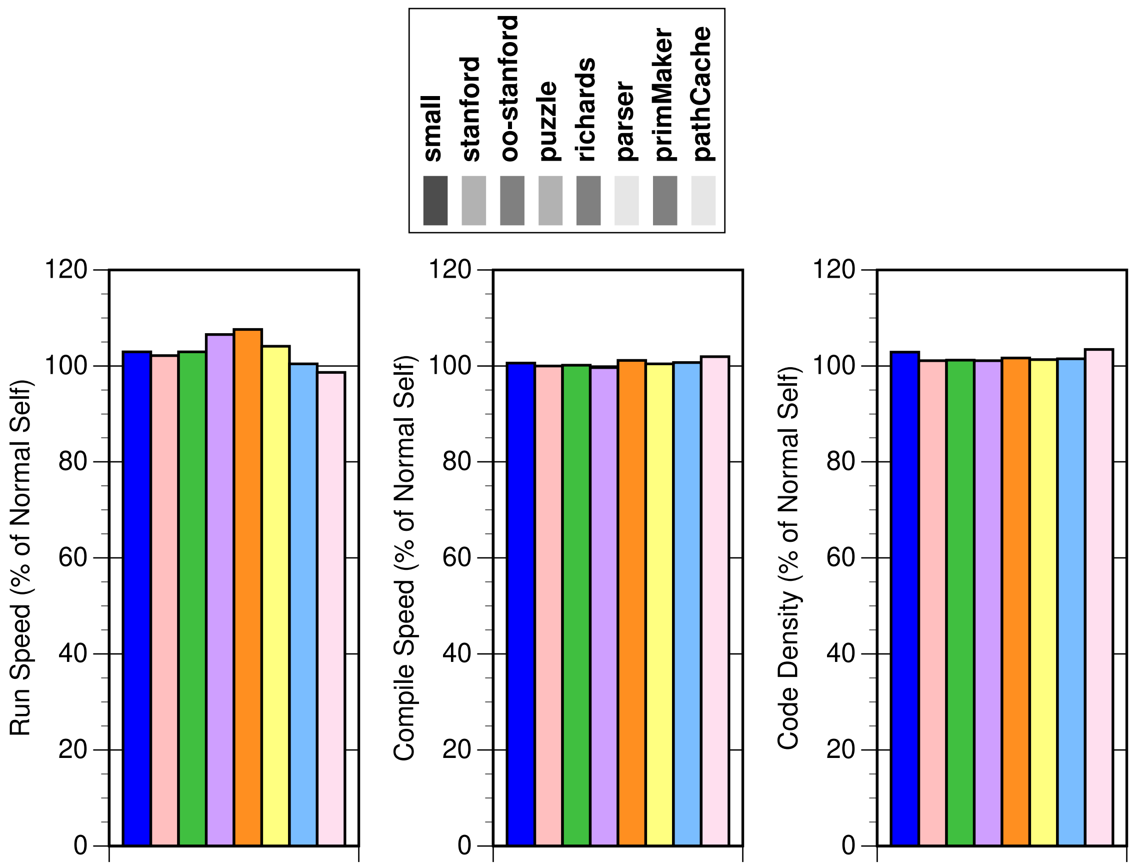
Interrupt checking imposes a moderate cost in execution time for most benchmarks. The more numerical benchmarks slow down by between 4% and 13% from interrupt checking at **`_Restart`**'s and a few percent from interrupt checking at method entries, while the larger, more object-oriented benchmarks slow down by a few percent from interrupt checking at method entries and virtually none from interrupt checking at **`_Restart`**'s. This difference reflects the fact that the smaller benchmarks contain more tight loops while the larger benchmarks contain more calls.

Generating code to handle interrupts imposes a fairly substantial compile-time cost, at least when compared to the other measured sources of overhead. Interrupt checking at **`_Restart`**'s is more costly than at calls, since it involves ensuring that all debugger-visible names are properly set up so that the debugger could display the virtual source-level call stack if invoked at the interrupt point. Supporting interrupts via polling imposes fairly significant compiled code space costs, with up to 35% extra compiled code generated for the smaller benchmarks. Again, interrupt checking at

_Restart's is more expensive than at calls since it may require extra code to support the debugger. It is unlikely, however, than either of these compile-time costs could be avoided by an alternative run-time mechanism for handling interrupts.

B.5.8 LRU Compiled Method Reclamation Support

Compiled methods are stored in a fixed-sized cache. Some compiled methods must be flushed from the cache to make room for new compiled methods when the cache is full. The system attempts to flush compiled methods that are unlikely to be used soon afterwards by keeping track of which compiled methods have been invoked recently, and flushing those compiled methods which have been used least recently. The compiler supports this *least-recently-used* (*LRU*) replacement strategy by generating extra code in the compiled method prologue to mark the compiled method as recently used, as described in section 8.2.3. This overhead would not exist in an implementation that either never threw away compiled code or replaced compiled methods in a different way (such as using *first-in first-out* (*FIFO*) replacement or LRU replacement with usage information computed via periodic interrupts). We measured the overhead for the current LRU implementation by disabling the marking code in the method prologue. The following charts display the costs of the current LRU compiled method reclamation support.



No LRU Compiled Method Reclamation Support

LRU support imposes a run-time cost of up to 8% over a system without this support. This cost is directly proportional to the call frequency of the program being measured. LRU reclamation support takes little additional compile time and little additional compiled code space.

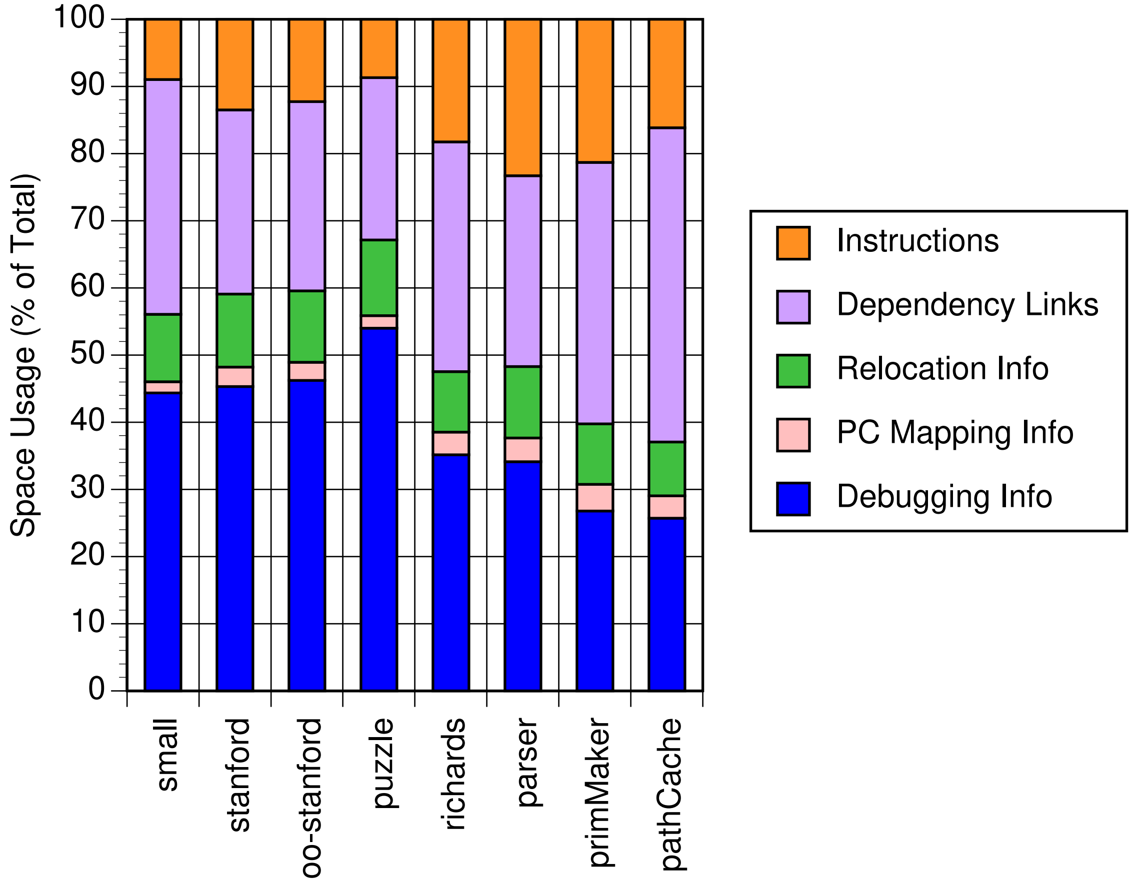
B.5.9 Summary of Remaining Sources of Overhead

Of the remaining sources of overhead we were able to measure, no single source of overhead imposes a significant execution speed cost. Only array bounds checking, type testing, overflow checking, interrupt checking, and LRU compiled method reclamation have a non-trivial execution time cost, and none incurs more than 10% overhead. Much of the remaining gap in performance between SELF and optimized C remains unaccounted for, however.

B.6 Additional Space Costs

The previous compiled code space efficiency measurements compared the number of machine instructions generated by the SELF compiler to the number of machine instructions generated by other configurations of SELF or by other language implementations. The SELF compiler generates additional information with compiled methods that takes up more space. These additional pieces of information include descriptions of inlined scopes and tables mapping between physical program counters and source-level byte code position. This information is used to reconstruct the virtual call stack from the physical call stack during debugging (as described in section 13.1), to support lazy compilation of uncommon branches, and to compile block methods that perform up-level accesses to lexically-enclosing stack frames. The compiler also generates dependency links, used for selective invalidation of compiled methods after programming changes (as described in section 13.2). Finally, the compiler generates information identifying the locations of all tagged object references embedded in compiled code and scope debugging information, to enable the system to update these pointers after a scavenge or garbage collection.

The following chart breaks down the space consumed by the output of the SELF compiler into the above categories of information, under the standard configuration.



This chart illustrates that machine instructions take only a relatively small fraction of the space consumed by compiler-generated data, around 10% of the space for the more numerical benchmarks and 20% of the space for the larger object-oriented programs. Scope debugging information and dependency links take up the lion's share of the space used by generated information, between 60% and 80% of the total generated space. Relocation information requires a roughly constant 10% of the total space. Program counter/byte code mappings are relatively concise, taking up less than 5% of the total space used.

Fortunately, not all of this information needs to remain in main memory at all times. Much of it can be paged out to virtual memory and brought into main memory only when needed. Compiled instructions need to be in main memory (for those methods in active use). Location information needs to be in main memory, since it will be accessed relatively frequently during scavenges.* Scope and p.c./byte code debugging information is needed only when compiling nested blocks and uncommon branch extensions and when debugging, so for methods not being debugged the scope information can be paged out relatively quickly after “warming up” the compiled code cache. Dependencies are only needed when programming and flushing invalid methods, so this space can be paged out when not in program development mode. Thus, for working, debugged programs most of the extra data generated by the compiler can be paged out of main memory, minimizing real memory requirements. Ultimately only the machine instructions and locations need be in main memory (and the latter only for garbage collections), thus keeping real memory space costs down to a fraction of the total virtual memory space costs.

* If all tagged object pointers embedded in a compiled method refer to tenured objects in old-space (objects not scanned as part of scavenging), this location information will only be needed during a full garbage collection and so normally can be paged out. As an optimization, the system could automatically tenure all objects reachable from compiled methods to allow the location information to be paged out immediately and to speed scavenges.