Do object-oriented languages need special hardware support?

Urs Hölzle
David Ungar†

Abstract: Previous studies have shown that object-oriented programs have different execution characteristics than procedural programs, and that special object-oriented hardware can improve performance. The results of these studies may no longer hold because compiler optimizations can remove a large fraction of the differences. Our measurements show that SELF programs are more similar to C programs than are C++ programs, even though SELF is much more radically object-oriented than C++ and thus should differ much more from C.

Furthermore, the benefit of tagged arithmetic instructions in the SPARC architecture (originally motivated by Smalltalk and Lisp implementations) appears to be small. Also, special hardware could hardly reduce message dispatch overhead since dispatch sequences are already very short. Two generic hardware features, instruction cache size and data cache write policy, have a much greater impact on performance.

Keywords: compilers, implementation, performance, computer architecture, object-oriented hardware

1. Introduction

How much can dedicated hardware improve the performance of object-oriented programs? Previous studies have indicated that special hardware can improve performance, and some implementations of object-oriented systems have relied heavily on hardware support. For example, the Xerox Dorado, for a long time the fastest Smalltalk implementation available, contained microcode support for large portions of the Smalltalk virtual machine [Deu83]. Ungar reported that the SOAR system would have been 26% slower without instructions for tagged arithmetic [Ung87]. Williams and Wolczko argued that software-controlled caching improves the performance and locality of object-oriented systems [WW90].

However, none of the systems previously studied employed compiler optimizations specifically aimed at reducing the overhead of message passing. Thus, the results of these studies may overstate the benefits of special hardware features. For this study, we used the SELF-93 implementation [Höl94]. Two factors make it a good vehicle for measuring the execution characteristics of object-oriented programs:

- SELF [US87], like Smalltalk-80 [GR83], is a very pure object-oriented language. All data are objects, and virtually every operation (e.g., instance variable access, integer addition, or control structures like `if` and `while`) involves message sends. The pure object-oriented language model also inspires a highly object-oriented programming style that makes frequent use of fine-grained abstractions. Thus, if object-oriented programs behaved differently than procedural programs, SELF programs should exhibit these differences in the extreme, much more than programs written in hybrid object-oriented languages like C++. Conversely, if SELF programs behaved just like C programs despite the extreme object-orientedness of the language (because the compiler can reduce the object-oriented source program to C-like object code), then it should also be possible to implement less extreme object-oriented languages in a similar way, so that their programs also behave like C programs.
- The SELF implementation employs several optimizations aimed specifically at reducing the overhead of dynamic dispatch ([CUL89], [HCU91], [CU91], [HU94]). With these optimizations, programs run several times faster than comparable Smalltalk programs (since the fastest Smalltalk system uses only a non-optimizing compiler) and

† Author's addresses: Urs Hölzle, Department of Computer Science, University of California, Santa Barbara, CA 93106, USA; urs@cs.ucsb.edu. David Ungar, Sun Microsystems Laboratories, MTV 29-116, Mountain View, CA 94043, USA; ungar@eng.sun.com.

Technical Report TRCS 94-21, Department of Computer Science, University of California, Santa Barbara, November 1994. UCSB CS reports are available via ftp from ftp.cs.ucsb.edu.

about half as fast as optimized C programs (for the benchmarks mentioned in [CU91] and [HU94]). Since the goal of this study was to determine if object-oriented programs still need special hardware support when compiled with state-of-the-art optimization techniques, the SELF-93 implementation was a good match.

We measured the behavior of several large SELF applications with an instruction-level tracing tool. The results of these measurements indicate that even an extremely object-oriented language like SELF can run efficiently on stock hardware (i.e., without special hardware support) since its execution characteristics are surprisingly similar to those of C programs. SELF's instruction mix is similar to that of the SPEC programs, and features like tagged arithmetic instructions or (hypothetical) instructions supporting message lookup all contribute less than 5% to performance. Overall, instruction and data caches can have a much larger impact on performance than the "OO" hardware features.

In the remainder of the paper, we compare SELF's execution characteristics against the C programs of the SPEC benchmark suite and against a suite of C++ programs. Then, we evaluate one feature present in the SPARC architecture (tagged arithmetic instructions) that has previously been shown to improve the performance of (unoptimized) object-oriented programs. We also investigate if hardware-assisted message lookup could improve performance. Finally, we examine the cache behavior of the programs.

2. Methods

The SELF-93 system compiles into native SPARC code, and thus all our data is for the SPARC V8 architecture [SPARC92]. The measurements were made with a combination of publicly available tools which were adapted for our purposes: the Shade tracing tool [CK93], the Dinero cache simulator [Hill87], and the SPA SPARC simulator [Irl91]. Our simulator models the Cypress CY7C601 chip, a scalar implementation of the SPARC architecture used in the SPARCstation-2 workstation. To reduce the simulation time, execution times were kept relatively short (25M instructions or less, see appendix). However, in a control experiment we ran three of the programs with larger inputs, resulting in execution times around 500M instructions. The resulting data were so similar that we believe the data presented here accurately reflect the behavior of the SELF system.

The SELF programs used for the study consist of large and medium-sized applications (see Table 1). The programs were written by several different programmers and represent a variety of programming styles and problem domains. Thus, we believe that they represent well the spectrum of typical SELF programs. Except for `Richards`, no program was written specifically as a benchmark, and most programs were in daily use at the time we took the measurements. As typical procedural programs, we chose the integer part of the SPEC89[†] benchmark suite which is widely used to evaluate the performance of workstations.[‡]

Benchmark		Source size[a]	Description
small benchmarks	DeltaBlue	500	two-way constraint solver developed at the University of Washington
	PrimMaker	1,100	program generating "glue" stubs for external primitives callable by SELF
	Richards	400	simple operating system simulator originally written in BCPL by Martin Richards
large benchmarks	CecilComp	11,500	Cecil-to-C compiler compiling the Fibonacci function (the compiler shares about 80% of its code with the interpreter, CecilInt)
	CecilInt	9,000	interpreter for the Cecil language running a short Cecil test program
	Mango	7,000	automatically generated lexer/parser for ANSI C, parsing a 700-line file
	Typeinf	8,600	type inferencer for SELF
	UI1	15,200	prototype user interface using animation techniques[b]
	UI3	4,000	experimental 3D user interface

Table 1. SELF benchmarks

[†] For the final paper, we hope to have SPECint92 data available.

[a] lines of code (excluding blank lines and comments).

[b] time for both UI1 and UI3 excludes the time spent in graphics primitives

To accurately characterize typical SELF and C workloads with our measurements, we use a suite of large and realistic programs. Unfortunately, the flip side of this emphasis on realistic applications is that such applications are available in one language only. Since the development of each of these programs involved a significant programming effort, it would be impractical to translate the programs into other languages. Furthermore, many of the programs (e.g., the user interfaces or the type inferencer) are too tightly coupled with their environment to be translated easily into other languages. Thus, the only two programs that allow direct comparisons between languages are the two smallest programs, `Richards` and `DeltaBlue`, which are available in both C++ and SELF. But, being relatively small and having few user-defined data structures, these two benchmarks may not by as indicative of real programs as the other applications.

In fact, when comparing programs written in two languages as radically different as SELF and C, one will always compare apples and oranges to a certain extent. Even when comparing the same program written in both languages, the two programs will not perform exactly the same computation. For example, SELF programs include generic arithmetic, overflow checks for integer addition, bounds checks for array accesses, run-time type checking, closures, and garbage collection. Thus, the comparison will never actually compare the same program; at best, the two programs are similar. One could try to maximize the similarity by writing "C-style" programs in SELF or "SELF-style" programs in C, but such contrived programs would no longer be representative of typical programs.

Since the goal of this study was to determine the level of hardware support needed by typical object-oriented programs, and to measure the difference between typical object-oriented programs and typical procedural programs, we felt that choosing large, typical programs from each domain would result in the most predictive results. In other words, realistic characterization of C and SELF workloads was deemed more important than head-to-head comparison of individual programs.

3. Instruction usage

When designing an architecture, one of the first tasks is to identify frequent operations and then to optimize their performance. Many previous studies have found the execution characteristics of object-oriented languages to be very different from C. For example, Smalltalk studies [Kra83][Ung87] have shown calls to be much more frequent than in other languages. Even for a hybrid language like C++ (which has C at its core and thus shouldn't behave too differently at runtime), significant differences were found. Table 2 shows data from a study by Calder et al. [CGZ94] which

	C++	SPECint92	ratio C++ / SPEC
basic block size	8.0	4.9	1.6
call/return frequency	4.6%	0.7%	6.7
instructions per conditional branch	15.9	6.4	2.5

Table 2. Differences between C++ and C (from [CGZ94])

measured the behavior of several large C++ applications and compared it to that of the SPECint92 suite and other C programs. The study used the GNU C and C++ compilers on the MIPS architecture. Even though C++ is similar in spirit to C, the differences in execution behavior are pronounced: for example, C++ executes almost 7 times more calls and executes less than half as many conditional branches.

Based on these numbers, one would expect a pure object-oriented language like SELF or Smalltalk to be much further away from C, because the language is so radically different from C. (Recall that, for example, even integer

‡ We did not use the floating-point SPEC Fortran benchmarks because we considered them too specific to the scientific computation domain to be usefully compared against other programs.

addition or `if` statements involve message sends in SELF.) However, execution behavior is a function not only of the source language characteristics but also of compiler technology, and our data show that the latter can make a big difference. Figure 1 shows the dynamic instruction usage of the four SPECint89 integer benchmarks (written in C) and the nine SELF programs we measured. The data is summarized using box plots† (the appendix contains detailed

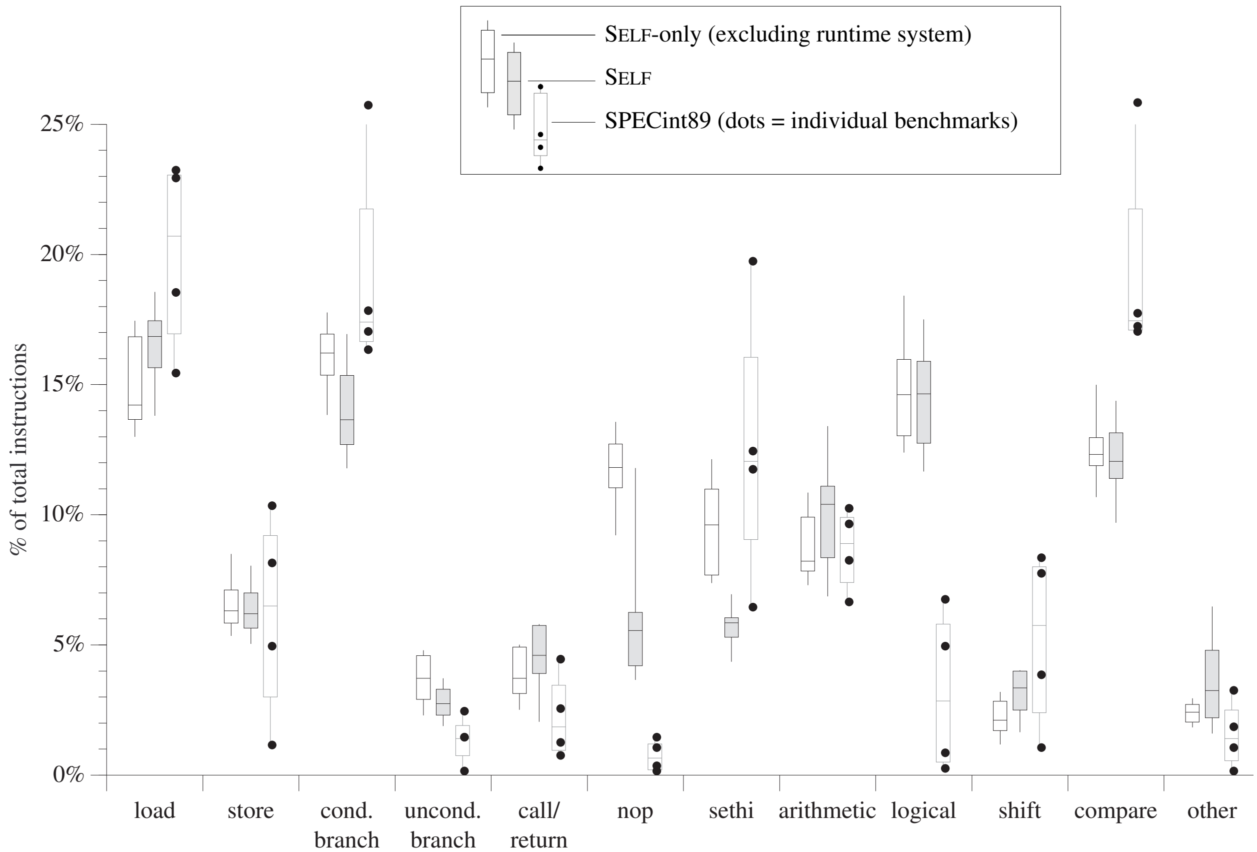

Figure 1. Dynamic instruction usage of SPEC integer benchmarks and SELF-93 benchmarks

data). The leftmost box in each category represents SELF-only, i.e., the execution of compiled SELF code, excluding any time spent in the runtime system (which is written in C++). The middle box in each category (filled gray) represents complete SELF programs (i.e., all user-mode instructions). We measured both since SELF programs often call routines in the runtime system, for example, to allocate objects or call functions defined in C libraries. Some programs spend one-third of their time in such routines, and separating out SELF-only ensures that our data is not biased by the execution behavior of non-SELF code. On the other hand, showing SELF-only could be misleading as well, since this data may not be representative of the instructions the processor actually executes.

Since there are only four benchmarks in the SPECint89 suite, the individual data points for C are shown directly, and the box plots (dotted) are given for reference only. The SPECint89 data are taken from Cmelik et al [Cme91].

Figure 1 reveals several interesting points:

1. Overall, there are few differences between the SPEC benchmarks and SELF. Often, the differences between the individual SPEC benchmarks are bigger than the difference between SELF and C (in the graph, the SPEC boxes are usually larger than the corresponding SELF boxes).

† A box plot summarizes a distribution by showing the median (horizontal line in the box), 25% / 75% percentiles (end of the box), and 5%/95% percentiles (vertical lines).

2. On average, SELF programs execute more no-ops, more unconditional branches, and more logical instructions. However, most these differences can be explained by the simple back end of the current SELF compiler, as explained in more detail below. Thus, they are an artifact of the current compiler back end and are not linked to the object-oriented nature of SELF.

3. SELF's basic block size is very similar to that the SPEC programs, 4.5 vs. 4.4 instructions (geometric means). However, calls and returns[†] are more frequent, occurring every 33 instructions (SELF-only) and 25 instructions (SELF) vs. every 57 instructions in SPEC89. Interestingly, SELF's runtime system (written in C++) has a higher call/return frequency than SELF code.

Before going into a more detailed comparison, we exclude two instruction categories that are distorted by deficiencies in the back end of the SELF-93 compiler. The back end does not fill delay slots (except in fixed code patterns), and thus SELF programs contain many no-ops (6.2% vs. 0.45% for C[‡]). Also, it does not optimize branch chains or rearrange code blocks to avoid unconditional branches; thus, the code contains more unconditional branches (2.7% vs. 0.8% in C). Since the presence of these extra instructions distorts the frequencies of other instruction categories, we exclude them in all future graphs in this section.

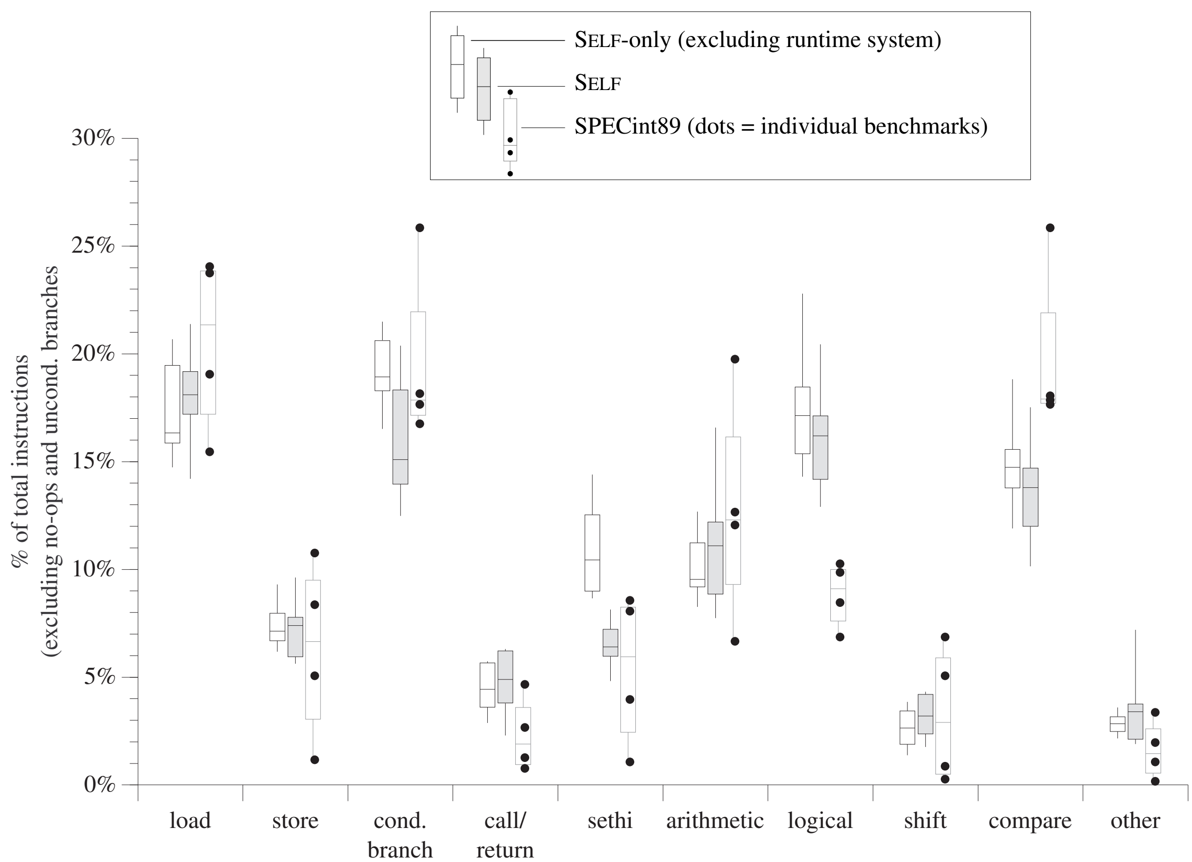

Figure 2. Dynamic instruction usage of SPEC89 integer benchmarks and SELF-93 benchmarks (adjusted)

Figure 2 shows the adjusted execution frequencies. Comparing SELF and SELF-only reveals that the runtime system does not influence the overall behavior much. The only two instruction groups showing a significant difference are

[†] Since SELF uses a non-standard calling convention and sometimes performs both a call and a (direct) jump to perform message dispatch, we are currently unable to measure the call frequency directly and use the call/return frequency instead. The call/return frequency is the frequency of all `jmpl` and `call` instructions combined. For C/C++, the call frequency is half the call/return frequency; for SELF, this is not the case (i.e., the call frequency is less than half of the call/return frequency since some calls involve both a `call` and a `jmpl`.)

[‡] The percentages are the geometric means of the frequencies.

conditional branches and `sethi` instructions. Conditional branches are more frequent in SELF-only than in SELF, but not unusually high compared to the SPEC programs. `sethi` instructions are used to load 32-bit constants; they are more frequent in compiled SELF code for two reasons. First, the values `true`, `false`, and `nil` are objects in SELF, represented by 32-bit code constants; in contrast, C programs can use the short immediates 0 and 1. Second, object types are represented by the address of the type descriptor, and the implementation of message dispatch compares the receiver's type against the expected type(s) [HCU91]. Since message dispatch is frequent, so are 32-bit constants.

Compared to the SPEC benchmarks, SELF shows few differences. Besides the differences in conditional branches and `sethi` instructions already mentioned above, only logical instructions and comparisons are markedly different. Logical instructions are more frequent in SELF for two reasons. First, the compiler's back end uses only a simple register allocator, and as a result generates unnecessary register moves.[†] (Move instructions account for roughly half of all logical instructions in SELF.) Second, SELF uses a tagged object representation with the two lower bits as the tag. Dispatch tests involving integers, as well as integer arithmetic operations, test their argument's tag using an `and` instruction; similarly, the runtime system (e.g., the garbage collector) often extracts an object's tag. Together, these `and` instructions account for about 25% of the logical instructions.

SELF executes fewer comparisons than the SPEC integer benchmarks. This result surprised us; we expected SELF to execute *more* comparisons since message dispatch involves comparisons and is quite frequent. If SELF used indirect function calls to implement message dispatch, one could explain the lower frequency of conditional branches with the object-oriented programming style which typically replaces `if` or `switch` statements with dynamic dispatch; Calder et al. observed this effect when comparing C++ programs to the SPEC C programs [CGZ94]. However, since the SELF implementation does not use indirect function calls, we cannot explain the difference in this way. It is possible that SELF's optimizer eliminates enough dispatch type tests to lower the overall frequency of comparisons.

Instruction category	Frequency relative to SPEC	Reasons for difference
call/return	1.5x higher (SELF-only)	different programming styles; possible compiler deficiencies in SELF-93
logical instructions	1.8x higher (SELF-only)	extra register moves (inferior back end); integer tag tests (`and` instructions)
sethi (load 32-bit constant)	1.5x higher	true/false/nil are 32-bit constants; message dispatch involves comparisons with 32-bit constants
comparison instructions	1.5x lower	see text

Table 3. Summary of main instruction frequency differences between SELF and SPECint89

Table 3 summarizes the main differences in instruction usage. Overall, there are few differences between SELF programs and the SPEC C programs, a surprising result considering how different the two languages are. What is even more surprising is that *SELF-93 is closer to GNU C than is GNU C++*. Figure 3 illustrates this point by summarizing the C++ data from [CGZ94] and our own measurements. In each category, SELF is closer to SPEC than is C++. For example, the basic block size in C++ is 1.6 times higher than SPEC, but in SELF it is only 1.1 times higher than SPEC.

Figure 4 shows the instruction mixes of the two benchmarks available in both C++ and SELF (`Richards` and `DeltaBlue`) in comparison to the SPECint median. In general, the data confirm the observations made above: SELF's instruction usage is very similar to SPECint (i.e., C) with the exceptions mentioned in Table 3, and C++ has a much higher call/return frequency (3-4 times higher than SELF) and significantly fewer conditional branches. For virtually all instruction categories, SELF is closer to the SPEC median than is C++.

In summary, our data show that execution behavior is more a function of compiler technology than of the source language characteristics. We believe that much of the difference between C++ (or other object-oriented languages) and C would disappear if compilers used OO-specific optimizations as well. Of course, until such systems are actu-

[†] Since moves are implemented using the `or` instruction, they are included as logical instructions.

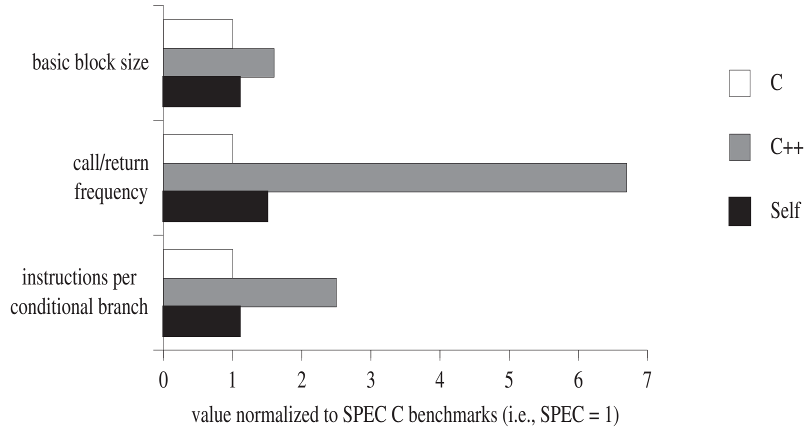

Figure 3. Differences between SELF/C++ and the SPEC C programs

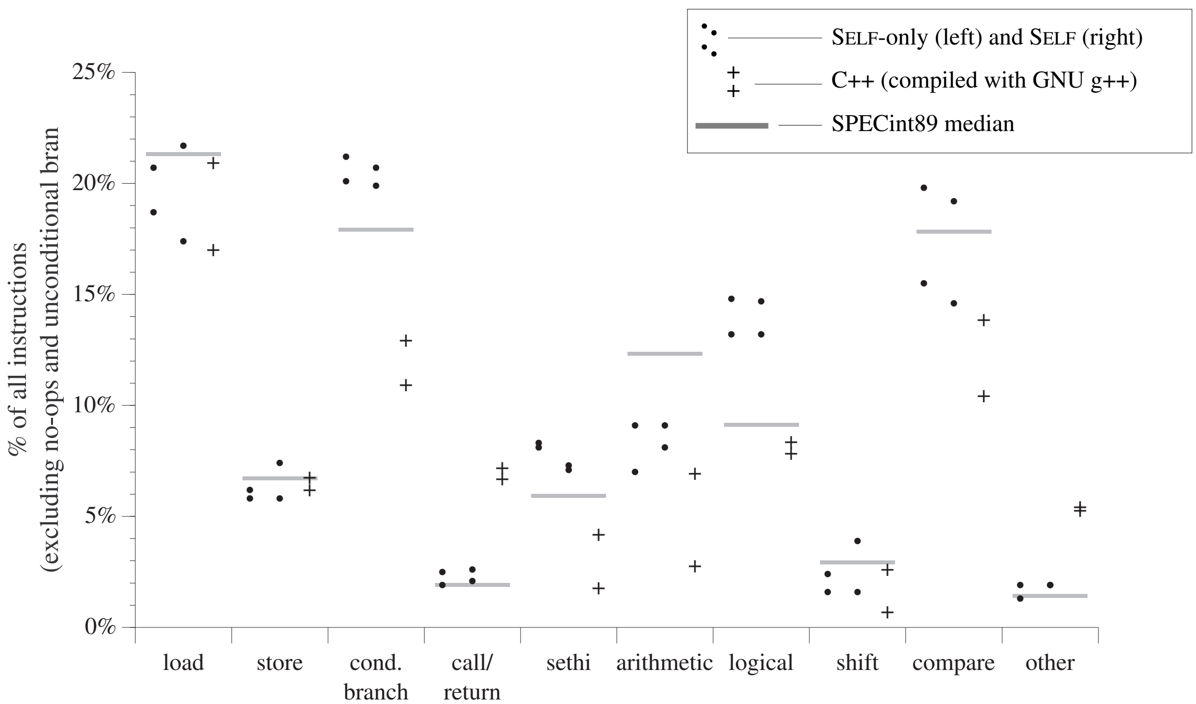

Figure 4. Instruction mix for SELF and C++ on `Richards` and `DeltaBlue` benchmarks

ally implemented, one cannot be certain that better optimization would indeed bring other object-oriented languages closer to C. However, in light of the data, any claims regarding the necessity of architectural support for an object-oriented language should be regarded with caution unless the system used for the study employs state-of-the-art optimization techniques, or unless such optimizations have been shown to be ineffective for that particular language. Unfortunately, all previous studies that we are aware of fall in this category (many were performed before OO-specific optimizations were developed).

In the next sections, we will examine the performance impact of hardware features that have been found to provide significant speedups when used with non-optimizing compilers. Our goal is to see whether these features still benefit performance in the presence of an optimizing compiler.

4. Hardware support for tagged arithmetic

Tagged integer addition and subtraction instructions are a unique SPARC feature that was motivated by the results of the SOAR project [SUH86]. The special instructions can perform an integer operation, a tag check (assuming a tag in the least significant two bits), and an overflow check in parallel [U+84]. Using a non-optimizing Smalltalk compiler, Ungar reported that SOAR Smalltalk would have been 26% slower without the tagged instructions [Ung87]. Similarly, Lisp machines have included hardware support for tagging [B+87], [SH87]. But how useful is hardware support in a system with an optimizing compiler? To answer this question, it is necessary to examine the code generated for integer addition (or subtraction) in SELF-93. Without special hardware, the expression `x + y`, when specialized for integers, is compiled into the following code:

```
if (x is integer) {                  // 3 instructions (test, cond. branch, unfilled delay slot)
   if (y is integer) {               // 2 instructions (fill delay slot with add)
         add(x, y, temp);            // 1 instruction
         if (overflow flag set) {    // 1 instruction (cond. branch)
               // handle error case: overflow or non-integer argument (code omitted for clarity)
         } else {
               result = temp;        // assign result (1 instruction, in delay slot of above branch)
         }
   } else {
         // handle non-integer case  (code omitted for clarity)
   }
} else {
   // handle non-integer case  (code omitted for clarity)
}
```

The first `if` implements the dispatch for the "+" message (in a pure object-oriented language, "`x + y`" means "send the + message to object `x`, passing object `y` as a parameter"). The code within the `if` is the inlined method for integer addition which calls the `IntAdd` primitive (also inlined) in the standard system.† Using the above code sequence, an integer addition takes 8 instructions. A better code sequence would fill delay slots, use only one tag test (by first ORing `x` and `y`), and eliminate the extra assignment, for a savings of 3 or 4 instructions. The comparison below assumes a 6-instruction sequence for integer addition or subtraction.‡

The ideal code sequence for addition, given the tagged addition instructions available in SPARC, is one instruction:

```
tagged_add_trapIfError(x, y, result);
```

This variant of the tagged add instruction (`taddcctv` in SPARC syntax) assigns the result register only if both `x` and `y` have integer tags and if the addition does not overflow; otherwise, the instruction traps. SELF-93 actually uses a non-trapping variant because the runtime system would need additional mechanisms to handle the traps and recompile the offending compiled code if necessary to avoid taking frequent expensive traps. However, to get an upper bound on the benefits of tagged instructions, we will assume that the ideal system would make maximal use of the tagged arithmetic support and execute all integer additions and subtractions in one cycle.

In addition to integer additions and subtractions, tagged instructions are also useful in integer comparisons†† and other integer tag tests (e.g., to verify that the index in an array indexing operation is an integer). For each integer tag test in these operations, we assume an overhead of 2 cycles (test + branch + unfilled delay slot – tagged add). There-

† The user could redefine the + method if she wanted.

‡ Replacing the whole integer addition sequence with a trapping tagged add requires the same optimization as folding the two tag tests into one does (using an OR): the compiler has to recognize that there are no instructions between the dispatch test (first integer tag test) and the addition (with the argument tag test). The fact that there are no instructions between the two tests is a result of the current definition of the SELF method for smallInteger addition. Since the definition can be changed by the user, the compiler cannot hardwire the code sequence. A peephole optimizer would suffice, but the current compiler does not perform peephole optimization. But if we assume a one-instruction sequence for the system using tagged instructions, we also have to assume a 6-instruction sequence (folded tag tests) for the system not using tagged instructions.

†† Comparisons differ from subtractions in that they do not have to check for arithmetic overflow.

fore, the number of cycles saved by tagged arithmetic instructions is 5 * (number_of_adds + number_of_subtracts) + 2 * number_of_other_integer_tag_tests.

program	% integer additions	% integer subtractions	% integer tag tests	estimated benefit of hardware support	benefit if delay slots are filled
CecilComp	0.5%	0.1%	1.8%	6.4%	3.5%
CecilInt	0.1%	0.1%	0.5%	1.7%	0.9%
DeltaBlue	0.6%	0.3%	3.7%	11.9%	6.4%
Mango	0.3%	0.2%	1.4%	5.1%	2.8%
PrimMaker	0.3%	0.1%	1.0%	4.0%	2.2%
Richards	0.4%	0.4%	2.3%	8.3%	4.5%
Typeinf	0.3%	0.2%	1.7%	5.9%	3.2%
UI1	0.6%	0.1%	2.4%	8.1%	4.4%
UI3	0.6%	0.3%	1.9%	8.2%	4.6%
Median	0.4%	0.2%	1.8%	6.4%	3.5%

Table 4. Frequency of integer operations and estimated benefit of tagged instructions

Table 4 shows the frequencies of integer operations in a system making maximum use of tagged instructions.[†] Integer arithmetic instructions are used rarely and represent only about 0.6% of all instructions; other integer tag tests are more frequent, with a median of 1.8%. Without hardware support for tagged integers, SELF would be 6% slower than a system making maximal use of tagged instructions.

However, the true savings are likely to be smaller since we made several simplifying assumptions that all tend to inflate the estimated benefits of tagged arithmetic instructions:

- We assumed that the compiler back end does not fill delay slots. Filled delay slots would save two instructions per addition and one instruction for each tag test, reducing the performance of tagged instructions from 6.4% to 3.5% (last row of Table 4).
- Counting instructions rather than cycles overestimates the savings because the code sequences are free of memory accesses and could probably execute at close to 1.0 CPI (cycles per instruction), whereas the overall CPI on a SPARCstation-2 is closer to 2 for the benchmark programs. Thus, the sequences replacing tagged instructions consume about 6% of all instruction but only about 3% of all cycles. The exact extent of this overestimation is of course machine-dependent, but we believe the non-tagged instructions could execute with a below-average CPI even on superscalar architectures with high branch penalties because the branches are extremely predictable (the integer type test will almost always succeed, and overflows are rare). Furthermore, existing superscalar SPARC implementations cannot execute the tagged instructions in parallel with other instructions [Sun90] (probably because of the potential trap), and so one tagged instruction consumes as much time as several other instructions (up to three for SuperSPARC [Sun90]).
- We assumed that the compiler has no type information on the arguments of integer additions or subtractions, and thus has to explicitly test both tags for all integer operations. This overestimates the cost of explicit tag checking since one argument may be a constant (e.g., when incrementing a loop index by 1) or otherwise known.

For these reasons, we consider 6% to be a generous upper bound of the benefits of tagged instruction support for the programs in our benchmark suite.[‡] For the SPARCstation-2, a more accurate upper bound is probably 2-3% of execution time. Thus, the instructions for tagged addition and subtraction do not improve SELF's performance much.

[†] The data include only additions/subtractions executed on behalf of source-level integer addition operations; other additions (e.g., in address arithmetic) are excluded since they operate on untagged values.

This result stands in marked contrast to Ungar's measurements showing that SOAR would have been 26% slower without instructions for tagged arithmetic [Ung87]. Why the big difference? By analyzing Ungar's data, we could find several reasons for the higher estimate:

- Ungar's data includes speedups from several tagged instructions (such as "load" and "load class") that are not needed in SELF. For example, the "load class" instruction is handled equally fast with a normal load instruction [Höl94]. Including only the tagged arithmetic instructions, Ungar's estimate would be 12%.
- The SOAR code sequences for integer arithmetic without tagged instructions are slowed down because SOAR does not have a "branch on overflow" instruction. With such an instruction, the code sequences would be shorter, reducing the estimated slowdown to 8%.
- Integer arithmetic is more frequent in SOAR than in SELF. In Ungar's benchmarks, arithmetic instructions represent 2.26% of all instructions vs. 0.64% in SELF. We do not know whether this difference is a result of compiler differences or of differences in the benchmarks.

Together, these factors explain the difference between our estimates and the SOAR estimates.

5. Hardware support for message lookup

Object-oriented languages replace traditional direct procedure calls with dynamically-dispatched "indirect" calls, where the target address of the call depends on the type of the receiver (and, in some languages, possibly on one or more arguments). Finding ("looking up") the correct target address was a bottleneck in early Smalltalk implementations [Kra83]. Since message dispatch is extremely frequent in pure object-oriented languages (recall that virtually every operation involves message sends), hardware support could possibly speed up execution.

To test this hypothesis, we measured the time spent in message dispatch (including all integer tag tests). Although the SELF compiler inlines many message sends, it still incurs some dispatch overhead even for inlined message sends to ensure that the correct inlined code sequence is executed. Thus, we include all such type tests in our overhead, as well as the time spent dispatching non-inlined message sends.† Message dispatch in SELF is implemented with polymorphic inline caching [HCU91], so that most sends incur only a small overhead (a load and a compare-and-branch sequence) in addition to the cost of a direct call.

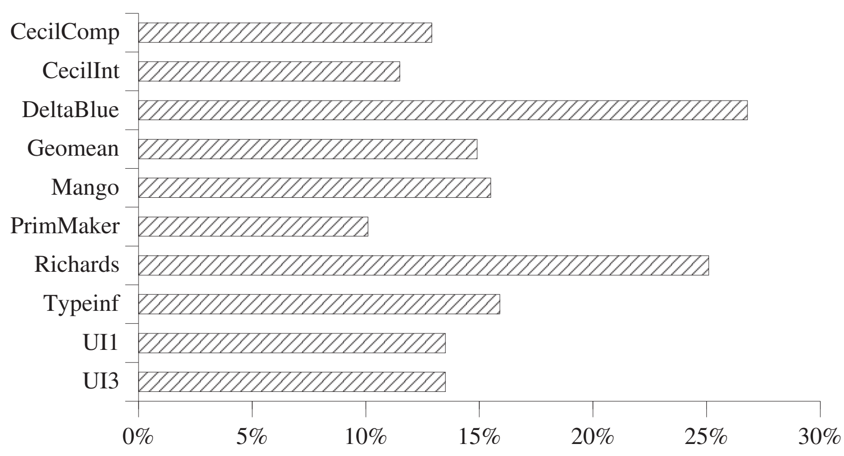

Figure 5. Total time spent in message dispatch

‡ Of course, programs with a higher frequency of integer arithmetic (such as the Stanford integer benchmarks) could benefit more from tagged arithmetic instructions. However, this class of programs is also more amenable to optimizations that eliminate type checks [CU91].

† The measurements include everything needed to resolve the target of a message send, but not the actual call/return overhead. All times include cache overhead, assuming the unified 64K direct-mapped cache of the SPARCstation-2.

Figure 5 shows that most programs spend less than 15% of their time in message dispatching, with the exception of the two smallest programs (`Richards` and `DeltaBlue`) whose overhead lies around 25%. At first sight, the relatively high overhead seems to indicate a possible opportunity for hardware support. However, the main reason for the overhead is the extreme frequency of dynamic dispatch. The median time per test is 8 cycles [Höl94]; for `Richards` and `DeltaBlue`, the time per test is even lower (4.5 and 3.4 cycles per test, respectively) since these programs send many messages to integers, so that integer tag tests dominate the dispatch overhead. Dispatch tests are very simple, consisting of two operations (loading the receiver's type and comparing it against a 32-bit constant) that require three to five instructions on a RISC processor.† Being so simple, dispatch tests cannot easily be improved with special hardware: the dispatch overhead consists of very many very simple and short tests, and thus there are no expensive computations that could be sped up with special hardware. The most promising way to reduce the overhead further probably is via compiler optimizations; the current compiler already eliminates more than 75% of all dispatch tests, i.e., less than one in four source-level message sends actually involves runtime dispatch overhead.

6. Instruction cache behavior

After examining the performance impact of architectural features targeted specifically at object-oriented languages, we will briefly discuss generic hardware features that influence performance. In particular, the size of the instruction cache has a large impact on SELF's performance. Figure 6 shows the instruction cache miss ratios for a range of direct-mapped caches; all caches use a 32-byte line size. Most of the SELF programs we measured are fairly large, consisting of several hundred Kbytes of code (see Table A-1 in the Appendix). For comparison, Figure 6 also includes `gcc`, the largest SPEC benchmark (about 900 Kbytes). Compared to `gcc`, the miss ratios of the SELF programs are significantly higher; for most cache sizes, the median miss ratio is 2-3 times higher than `gcc`'s miss ratio.

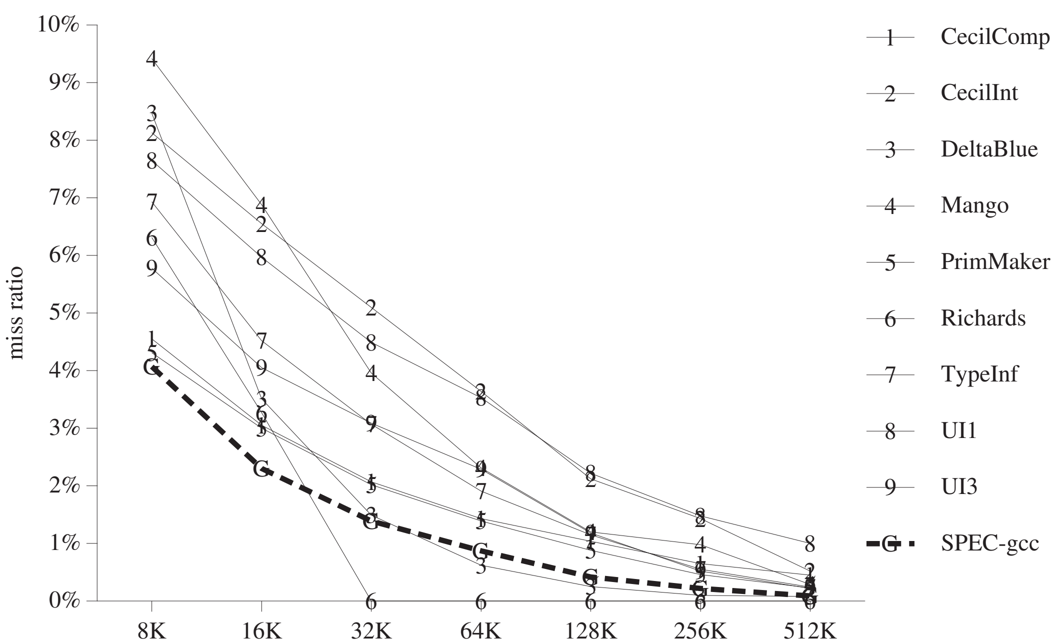

Figure 6. Instruction cache miss ratios of SELF programs (direct-mapped cache, 32-byte lines)

† With a slightly different run-time organization (using 13-bit type numbers instead of 32-bit type pointers) a test could be reduced to three instructions (load, compare with immediate, and branch). SELF-93 does not use type numbers because they complicate the runtime system (in particular, garbage collection). We considered the potential speedup too small to warrant introducing these complications.

For caches up to 32K, the misses are dominated by capacity misses; a 4-way associative cache would reduce misses by a factor of 1.4 (see Table A-3 in the appendix). For larger caches, that factor increases, peaking at 4.5 for a 256K cache, indicating that conflict misses play a bigger role in larger caches. However, conflict misses cannot explain the higher cache misses: even with a four-way associative cache, the median miss ratio is twice as high as `gcc`'s (data not shown). Apparently, the temporal locality of SELF programs is significantly worse than that of `gcc`. One reason might be that object-oriented programs have more distributed control and lack the tight loops that would enable good cache behavior even for large programs, or that `gcc` has an unusually good cache behavior for a program of its size. A more detailed study is beyond the scope of this paper but appears a promising area for future work.

The high instruction cache misses have a substantial performance impact. With a 64K direct-mapped instruction cache with a 26-cycle miss penalty (the miss penalty of a SPARCstation-2), SELF runs 32% slower than it would with an infinitely large cache. Thus, the size of the instruction cache can have a larger influence on performance than all the object-oriented hardware features previously discussed.

7. Data cache behavior

Figure 7 shows the data cache read miss ratios for direct-mapped caches with 32-byte lines, write-allocate, and subblock placement. The write-allocate policy has been shown to be beneficial for programs with intensive heap allocation; we will discuss it in more detail below. We assume that there is no cost associated with write misses, i.e., that write buffers can absorb almost all writes (see [DTM94], [Rei93], or [Rei94] for justifications of this assumption).

Even though many SELF programs allocate objects at a rate of about 1 Mbyte/s, the data cache performance is good. For example, the median miss ratio with a 64K cache is 2.4%, which would lead to an overhead of 6% on a SPARCstation-2-like memory system (cache miss time = 26 cycles). Even with an 8K data cache, the median overhead would still be less than 17%. This data is consistent with that of Diwan et al [DTM94] who have measured allocation-intensive ML programs and found very low data cache overheads for the same cache organization (write-allocate, subblock placement). Similar results have also been reported by Reinhold for Scheme programs [Rei93], by Jouppi for the SPEC benchmark suite [Jou93], and by Koopman et al for combinator graph reduction [KLS92]. Such low data cache overheads leave little room for improvement through special cache features (e.g., [PS89], [WW90]).

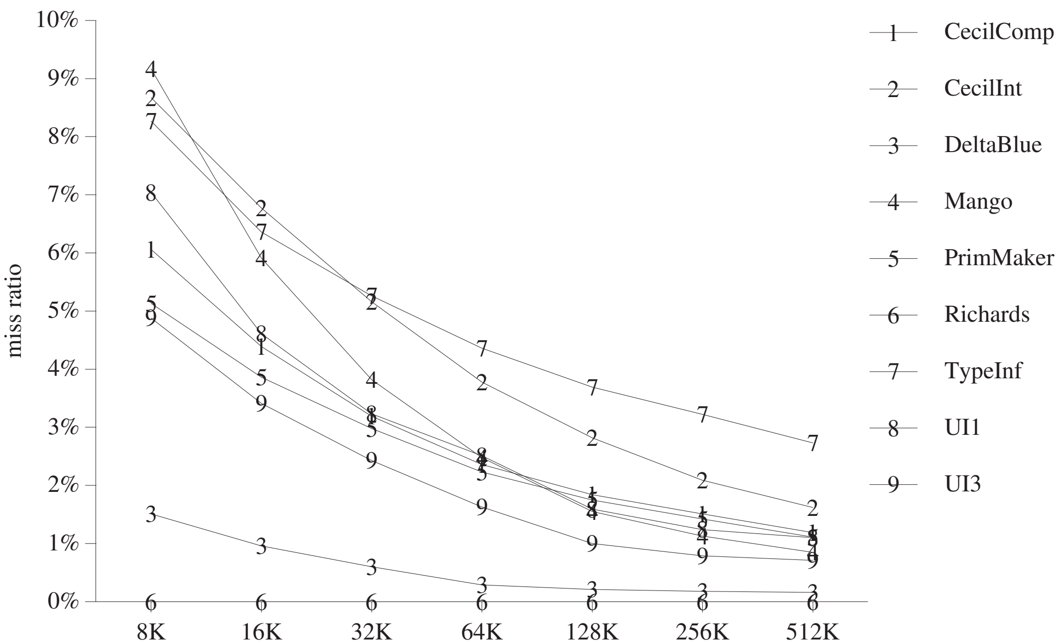

Figure 7. Data cache read misses (direct-mapped, 32-byte lines, write-allocate with subblock placement)

Diwan et al [DTM94] also observed that the data cache overhead of ML programs increased substantially with a write-noallocate policy, i.e., with a cache that does *not* allocate a cache line on a write miss. This is also true for SELF (see Figure 8). For a 32K cache, write-noallocate increases cache overhead by a median factor of 1.4, and this factor

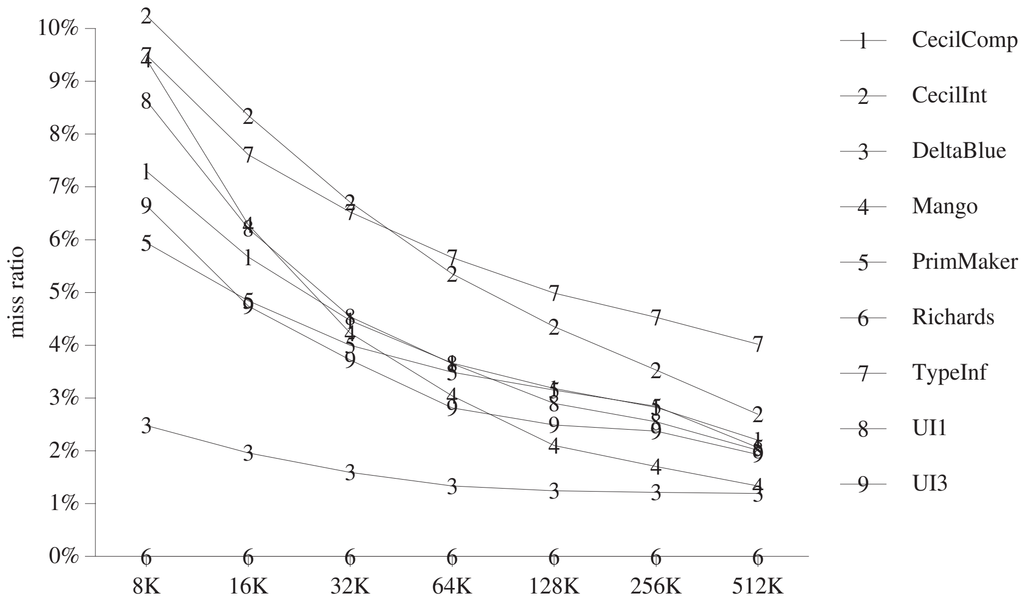

Figure 8. Data read misses (direct-mapped, 32-byte lines, write-noallocate)

increases with larger caches up to a factor of 1.8 for a 512K cache.

For write misses, the impact of the write policy is much higher: while write miss ratios are between 2% and 9% for all programs for write-allocate caches between 8K and 512K, the write miss ratios become truly staggering with write-noallocate caches. Miss ratios range from 20% to over 70% (!) for virtually all cache sizes and show very little

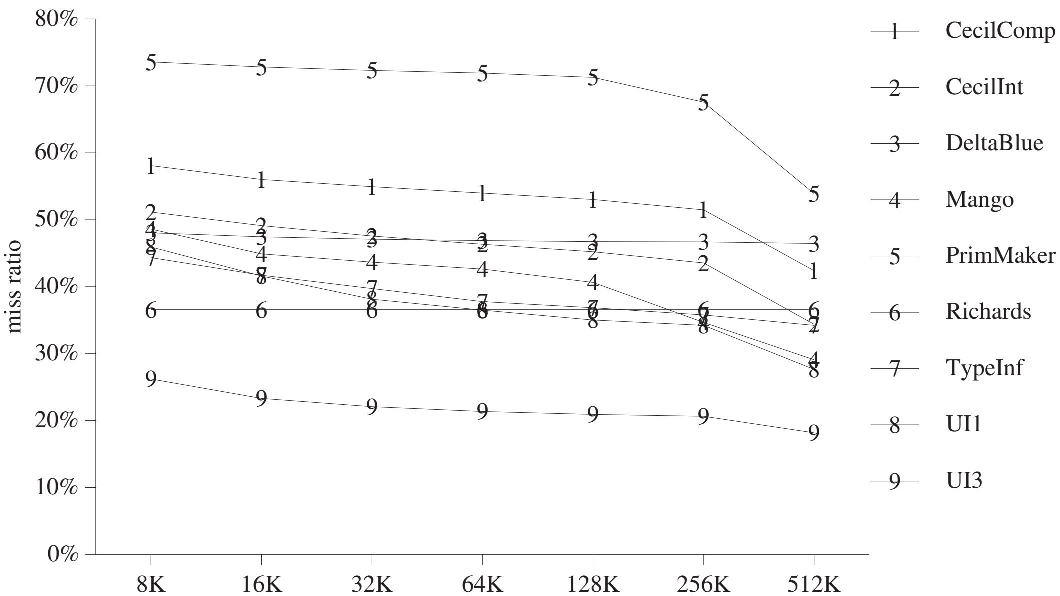

Figure 9. Write miss ratio (direct-mapped, 32-byte lines, write-noallocate)

improvement with larger caches (see Figure 9). Although we assume that there is no cost associated with write misses, it is still interesting that miss ratios are so high. The main reason is that objects are allocated consecutively in an allocation area. Between two garbage collections, the allocation pointer sweeps through the entire allocation space

(400 Kbytes in SELF). Since the fields of newly allocated objects are initialized before they are read, each initializing store will cause a cache miss in a system without write-allocate.

Wilson et al. [WLM92] argue that this allocation behavior will result in especially high cache overheads for direct-mapped caches since the allocation pointer will sweep through the entire cache and evict every single cache line. Therefore, they argue that caches should be associative, and that the size of creation space should be smaller than the cache size. Our data does not support this argument—we cannot detect a significant improvement when going from a 256K cache to a 512K cache, as predicted by Wilson et al. (the allocation area is 400Kbytes). To test Wilson's hypothesis further, we measured versions of SELF using 50Kbyte and 200Kbyte creation spaces. In both cases, overall execution time increased (by 50% and 10%, respectively) because of higher garbage collection overhead. With a 50Kbyte creation space, scavenges become eight times more frequent but not eight times less expensive (per scavenge) since the root processing time remains constant; also, object survival rates increase because of the shorter time intervals between scavenges. The increased scavenging activity[†] pollutes the data cache, so that the data read misses actually *increase* relative to the standard system for almost all benchmarks and cache sizes. For example, for caches between 64K and 256K, data read misses are 50-100% higher when using a 200Kbyte allocation area.

8. Future work

The results presented here could be extended in several ways. In addition to comparing single-instruction frequencies, comparing the relative frequencies of instruction groups (i.e., instruction pairs or triples) might reveal additional differences or similarities between implementations. Further work is needed to explore the impact of superscalar and out-of-order instruction issue. Of particular interest is the predictability of branches since it has a large impact on superscalar efficiency.

Further research is also needed to determine the cause of the higher instruction cache miss ratios observed in section 6. Since the performance impact of these misses is relatively high, software optimizations to reduce them may be worth investigating. Finally, even though cache-level data locality is good, page-level locality may be poor [WWT87] and thus needs to be studied.

9. Conclusions

We have measured the execution behavior of a pure object-oriented language (SELF) whose implementation uses compiler optimizations specifically targeted at object-oriented languages. SELF is arguably one of the purest object-oriented languages; if any object-oriented language behaved differently than C, SELF would be a prime candidate.

Surprisingly, compiler optimizations can almost entirely eliminate the large semantic difference between a pure object-oriented language and C. As a result, we found little opportunity for object-oriented hardware; overall, compiled SELF looks remarkably like C. In particular, we found that

- SELF's instruction mix is very similar to that of the SPECint89 suite, with few differences except for a 50% higher call/return rate, more logical instructions (partially caused by a simple compiler back end), and fewer comparisons. The basic block size is virtually identical to SPEC.
- In several respects, SELF programs are much more similar to C than are C++ programs compiled without OO-specific optimizations. For example, C++ programs have a much higher call/return frequency and significantly fewer conditional branches. The fact that compiled SELF is more similar to compiled C despite its pure language model is a strong indication that compiler optimizations influence run-time behavior more than language features.
- Even when using optimistic assumptions, SPARC's instructions for tagged arithmetic improve performance by at most 6%; more realistically, the improvement is 2-3%.
- Despite the extreme frequency of dynamic dispatch, hardware support for message lookup is not likely to help, because the dispatch overhead consists of very many very short tests which could hardly be optimized with extra hardware.

[†] In particular, processing the remembered set containing old-to-new references virtually flushes the cache in our implementation since it scans through a byte array that can exceed 32 Kbytes for the large heaps used by our benchmark programs.

- The data cache behavior of the test programs is good even with small caches, and thus there is little need for special object-oriented caches. Write-allocate caches with subblock placement reduce read miss ratios by up to a factor of two, and write miss ratios by a factor of ten. These findings are consistent with other work for non-object oriented languages ([KLS92], [Jou93], [Rei93], [DTM94]).
- Instruction cache size significantly impacts performance. For example, doubling the instruction cache from 32K to 64K improves performance by 15% on a SPARCstation-2. This improvement is higher than that of any OO-specific architectural feature we considered.

Our results contradict (and, we believe, supercede) those of earlier studies. While the execution behavior of object-oriented systems *without* optimizing compilers may differ markedly from SPEC programs, it appears that OO-specific optimizations can eliminate most of these differences even for a radically pure object-oriented language. In light of our data, any claims regarding the necessity of architectural support for an object-oriented language should be regarded with caution unless the system used for the study employs state-of-the-art optimization techniques, or unless such optimizations have been shown to be ineffective for that particular language.

Acknowledgments: We would like to thank everybody who commented on earlier drafts: Bob Cmelik, Karel Driesen, Bill Joy, David Keppel, Randy Smith, and Mario Wolczko.

References

[B+87] C. Baker et al. The Symbolics Ivory processor: a 40 bit tagged architecture Lisp microprocessor. *Proceedings of the 1987 IEEE International Conference on Computer Design*, p. 512-15, Rye Brook, NY, October 1987.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June 1991.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Conference Proceedings*, Phoenix, AZ, October 1991.

[Cme91] Robert F. Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly. An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks. *ASPLOS IV*, Santa Clara, CA, April 1991.

[CK93] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Sun Microsystems Laboratories, Technical Report SMLI TR-93-12, 1993. Also published as Technical Report CSE-TR 93-06-06, University of Washington, 1993.

[Deu83] L. Peter Deutsch, The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture. In [Kra83].

[DTM94] Amer Diwan, David Tarditi, and Eliot Moss. Memory Subsystem Performance of Programs with Intensive Heap Allocation. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 1-14, January 1994.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Second Edition, Addison-Wesley, Reading, MA, 1985

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, Geneva, 1991. Published as *Springer Verlag Lecture Notes in Computer Science 512*, Springer Verlag, Berlin, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94 Conference Proceedings*, pp. 326-335, Orlando, FL, June 1994. Published as *SIGPLAN Notices 29(6)*, June 1994.

[Höl94] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* Ph.D. Thesis, Technical Report STAN-CS-TR-94-1520, Department of Computer Science, Stanford University, 1994.

[Hill87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Technical Report UCB/CSD 87/381, Computer Science Division, University of California, Berkeley, November 1987.

[Irl91] Gordon Irlam. SPA—SPARC analyzer tool set. Available via ftp from cs.adelaide.edu.au, 1991.

[Jou93] Norm Jouppi. Cache Write Policies and Performance. In I*SCA'20 Conference Proceedings*, pp. 191-201, San Diego, CA, 1993. Published as *Computer Architecture News 21(2)*, May 1993.

[Kra83] Glenn Krasner, ed., *Smalltalk-80: Bits of History and Words of Advice.* Addison-Wesley, Reading, MA, 1983.

[KLS92] Philip Koopman, Peter Lee, and Daniel Siewiorek. Cache behavior of combinator graph reduction. T*OPLAS 14(2)*: 265-297, 1992.

[Pat85] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM 28 (1)*: 8-21, January 1985.

[PS89] Chih-Jui Peng and Gurindar Sohi. *Cache memory design considerations to support languages with dynamic heap allocation*. Technical Report 860, University of Wisconsin, July 1989.

[PH93] D. N. Pnevmatikatos and M. D. Hill. Cache Performance of the Integer SPEC Benchmarks on a RISC. *Computer Architecture News 18 (2)*: 53-68.

[Rei93] Mark Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. Technical report MIT/LCS/TR-581 (Ph.D. thesis), Massachusetts Institute of Technology, September 1993.

[Rei94] Mark Reinhold. Cache Performance of Garbage-Collected Programs. In *PLDI '94 Conference Proceedings*, pp. 206-217, Orlando, FL, June 1994. Published as *SIGPLAN Notices 29(6)*, June 1994.

[SUH86] A. Dain Samples, David Ungar, and Paul Hilfinger. SOAR: Smalltalk Without Bytecodes. OOPSLA '86 Conference Proceedings, pp. 107-118, Portland, OR, September 1986. Published as SIGPLAN Notices 21(11), November 1986.

[SPARC92] SPARC International. *The SPARC Architecture Manual (Version 8)*. Prentice Hall, NJ, 1992.

[SH87] Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and Software Approaches. In *ASPLOS II Conference Proceedings*, October 1987.

[Sun90] Sun Microsystems. *The Viking Microprocessor (T.I. TMS S390Z50) User Documentation*. Part No. 800-4510-02, November 1990.

[U+84] David Ungar, Ricki Blau, Peter Foley, A. Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. 11th Annual Symposium on Computer Architecture, Ann Arbor, Michigan, June 1984.

[Ung87] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, Cambridge, MA, 1987.

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

[WWT87] Ifor Williams, Mario Wolczko, and Trevor Hopkins. Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy. In *ECOOP '87 Conference Proceedings*, Special Issue of BIGRE, pp. 87-96, Paris, France, June 1987.

[WW90] Ifor Williams and Mario Wolczko. An Object-Based Memory Architecture. In *Proc. 4th Intl. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, September 1990.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching Considerations for Generational Garbage Collection. In *LFP'92 proceedings*, p. 32-42, San Francisco, June 1992.

Appendix A. Detailed data

Benchmark		static code size (bytes)[a]	dynamic code size (bytes)[b]	Kbytes allocated	dynamic data size (bytes)[c]
small benchmarks	DeltaBlue	39,960	31,020	332	336,444
	PrimMaker	161,640	128,292	2,435	877,512
	Richards	11,132	10,492	1	1,588
large benchmarks	CecilComp	922,452	547,684	1,283	1,023,996
	CecilInt	268,896	183,280	1,261	773,616
	Mango	239,020	223,056	1,078	958,488
	Typeinf	322,500	128,292	680	1,594,536
	UI1	954,812	625,808	542	746,784
	UI3	164,640	97,192	485	726,728

Table A-1: Benchmark data and instruction size

[a] excluding code in the runtime system
[b] unique instructions referenced (including runtime system)
[c] unique memory locations referenced by load and store instructions (including references by runtime system)

	taddcc	tsubcc	other tag tests	total instructions[a]
CecilComp	90,978	18,921	339,713	19,063,103
CecilInt	14,329	7,314	69,507	14,420,873
DeltaBlue	30,725	16,517	194,005	5,236,611
Mango	82,388	44,062	407,302	28,537,483
PrimMaker	85,040	16,539	261,769	25,742,677
Richards	69,767	61,853	397,965	17,500,114
Typeinf	45,956	22,182	218,912	13,259,725
UI1	54,687	9,225	222,917	9,397,411
UI3	64,637	27,176	195,422	10,324,703

Table A-2: Frequency of integer arithmetic and integer tag tests

[a] assuming optimal use of tagged instructions

cache size	direct-mapped misses / 4-way misses
8K	1.3
16K	1.35
32K	1.46
64K	1.76
128K	2.27
256K	4.47
512K	3.44

Table A-3: Median increase in miss ratio of direct-mapped vs. 4-way associative cache