INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Sather Language Design and Performance Evaluation[*]

Chu-Cheow Lim[†] Andreas Stolcke[‡]

TR-91-034

May 1991

Abstract

Sather is an object-oriented language recently designed and implemented at the International Computer Science Institute in Berkeley. It compiles into C and is intended to allow development of object-oriented, reusable software while retaining C's efficiency and portability. We investigate to what extent these goals were met through a comparative performance study and analysis of Sather and C programs on a RISC machine. Several language design decisions in Sather are motivated by the goal of efficient compilation to standard architectures. We evaluate the reasoning behind these decisions, using instruction set usage statistics, cache simulations, and other data collected by instrumented Sather-generated code.

We conclude that while Sather users still pay a moderate overhead for programming convenience (in both run time and memory usage) the overall CPU and memory usage profiles of Sather programs are virtually identical to those of comparable C programs. Our analysis also shows that each of the choices made in Sather design and implementation is well justified by a distinctive performance advantage. It seems, then, that Sather proves the feasibility of its own design goal of making object-oriented programming efficient on standard architectures using a combination of judicious language design and efficient implementation.

[*]Computing facilities used in this study are funded by National Science Foundation Infrastructure Grant number CDA-8722788 to U.C. Berkeley.

[†]ICSI and Computer Science Division, U.C. Berkeley. E-mail `clim@icsi.berkeley.edu`.

[‡]ICSI and Computer Science Division, U.C. Berkeley. E-mail `stolcke@icsi.berkeley.edu`. Author is supported by an IBM Graduate Fellowship.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Sather and Object-Oriented programming

Sather[3] is an object-oriented language designed and implemented recently at the International Computer Science Institute in Berkeley. Why another—in particular, object-oriented—language, one might ask?

The answer is that none of the existing object-oriented languages available seem to provide all of the major features Sather was designed for, namely

Code Reusability. The view of software components was proposed in the sixties, but has never been fully realized. Because object-oriented programming emphasizes the structuring of programs around *classes*[1] of objects, if the classes are well-encapsulated, a programmer simply needs to write "glue" code that ties together pre-written software components, allowing greater code re-utilization and productivity. Among today's languages, *Eiffel* seems to come closest to this goal. Therefore, Sather was strongly inspired by Eiffel[2], retaining the crucial features of class parametrization, multiple inheritance, modular class definitions, and a clean class interface.

Code Simplicity. Why, then, not simply adopt Eiffel? The problem with Eiffel is that it has a lot of features which are of only theoretical interest to most programmers. These include constructs which assert the pre- and post-conditions of routines and complex class inheritance rules. Hence, Sather was designed as a simplified variant of Eiffel. It focuses on providing features that allow programmers to write efficient, reusable code. In the environment Sather was developed in, runtime is still generally the major criterion for software suitability, and Sather code would only be used (much less reused) if it can afford performance levels comparable to traditional procedural languages.

Efficiency. Another reason for not choosing Eiffel is that the dispatching mechanism in the language makes Eiffel programs inherently slow to execute on standard architectures. Why use not C++[5], then, which has compilers that produce efficient code? This brings us back to the first point: C++ does not have a clean syntax and lacks constructs such as class parametrization which are essential to writing reusable code.

1.2 Historical Perspective

Sather is not the first attempt at trying to make object-oriented programs efficient. An example of implementing efficient object-oriented programming environment is the SOAR (Smalltalk On A RISC) project [6]. SOAR, however, tries to bridge the semantic gap between language and hardware from exactly the opposite side to Sather. More specifically, SOAR tries to gain efficiency of Smalltalk programs using an architectural approach. Starting with a RISC architecture, the aim was to add a certain number of architectural features, each of which would significantly improve certain aspects of the performance of Smalltalk programs. The disadvantages of such an approach include:

- The architectural features are closely tuned to the needs of a specific language (Smalltalk). It is not clear that the highly specialized solutions in SOAR can be equally useful in other object-oriented languages.
- It is difficult for Smalltalk systems implemented on SOAR to immediately reap the benefits of faster RISC architectures, since that would mean starting another hardware design cycle.

[1]In subsequent discussion, we use the terms *class* and *type* interchangeably.

Since RISC architectures—including RISC compiler technology—have further improved, it is appropriate to re-evaluate the performance of object-oriented systems on standard architectures, and see how efficiency can be obtained using an approach that does not rely on special-purpose hardware as SOAR.

In contrast to SOAR, Sather attempts to make object-oriented programs efficient by eliminating as many of the inefficiencies inherent in object-oriented language semantics at compile time. In many respects the language is designed to help the compilation process.

The language does not try to create the illusion that all language constructs are equally efficient, and asks the programmer to sacrifice some semantic generality and conceptual elegance to avoid significant performance penalties. Since Sather was developed out of concrete programming experience and demands, its designers feel safe to say that none of the restrictions incorporated in the language present a serious limitation, but are natural matches for real world programming needs.

1.3 Important Sather Language Features

In this section, we discuss two features of the Sather language that are designed to allow efficient code to be generated by providing crucial information about objects at compile time.

1.3.1 Explicit Dispatching

Most object-oriented languages have very general semantics for object dispatching. Consider the following piece of code:

```
x:ANIMAL;
x.eat;
```

In Eiffel, for example, `x` during execution may refer to an object in the class `ANIMAL` or in any of `ANIMAL`'s descendent class such as `MAMMAL`, `REPTILE`, etc. Furthermore, each class `ANIMAL`, `MAMMAL` or `REPTILE` may have a different `eat` routine. The only way to determine the correct `eat` routine to be executed, is to associate a tag with the object referred to by `x`. The object `x` is dispatched to the correct routine using the tag during program execution.

This incurs extra costs during execution time, because every operation on an object has to be dispatched. Sather takes the view that the programmer will have a good idea of the types of objects referred to by `x`. Often, the programmer knows that `x` can only refer to exactly one type of object. Then why incur the extra execution cost of dispatching? Therefore, in Sather, there is a distinction between dispatched and non-dispatched types. In the following piece of code (same as above),

```
x:ANIMAL;
x.eat;
```

`x` has a non-dispatched type and hence its type `ANIMAL` is known exactly during compile-time. The Sather compiler can then generate code to call `ANIMAL`'s `eat` routine, instead of generating dispatching code to determine the correct `eat` routine during actual program execution. There is no need to perform any type-checking during execution to check that `x` is of the correct type.

On the other hand, if the programmer decides that `x` may refer to any object from more than one class, then he/she can explicitly request the object to be dispatched as in the following code:[2]

```
x:$ANIMAL;
x.eat;
```

[2]The dollar sign syntax is meant to remind the programmer of the more costly nature of dynamic dispatching.

In this example, `x` has a dispatched type, and general code is generated so that `x` can be dispatched to the correct `eat` routine during execution.[3] (Later, we will discuss how the Sather implementation tries to make dispatching efficient.)

Hence Sather trades complete generality for simple restrictions which will improve code efficiency.

1.3.2 Separate Class Hierarchies for Basic Types

Basic types in Sather refer to the classes `INT`, `CHAR`, `BOOL`, `DOUBLE` and `REAL`. These are data types which are generally supported at the machine level with efficient operations. Some object-oriented languages (e.g. Eiffel, Smalltalk) have a top-most class which is inherited by all other classes (including basic classes such as `INT`). Thus we may have a class `INT_OR_DOUBLE` which is a descendent of both `INT` and `DOUBLE`. The following questions then arise:

- How much space is allocated for an object in the class `INT_OR_DOUBLE`?
- If we have

  ```
  x:INT_OR_DOUBLE;
  x := x + x;
  ```

  we only know at execution time whether the *add* routine refers to an integer add-operation or a double-precision floating-point add-operation. Hence, an extra tag is needed and this makes the addition operation expensive even though integer addition is supported efficiently at the machine level (and a lot of architectures have floating point units for floating-point operations).

To solve both problems, the Sather language differs from other object-oriented languages in that there is no top-most class which is the ancestor of all classes. There are several distinct hierarchies:

Basic Classes. Each basic class `INT, CHAR, BOOL, REAL` and `DOUBLE` has its own distinct inheritance hierarchy. Hence it is a compile-time error to define a class such as `INT_OR_DOUBLE` which inherits from both `INT` and `DOUBLE`. This simplifies the implementation and allows for efficient code to be generated as illustrated in the following example.

Suppose if we have a class `INT_HASH` which inherits from `INT`, and the following piece of code:

```
x:INT_HASH;
x := x + 1;
result := x.compute_hash_function;
```

The distinct hierarchies allow the compiler to treat `x` exactly as an integer with the additional property that it has an associted hash function. No extra space is needed to allocate any tag for `x` and there is no need to perform any type-checking during execution to decide whether `+` refers to an integer or a floating-point addition.

Non-Basic Classes. All other classes form a separate inheritance DAG with a top-class class called `OB`. During execution, any object from a non-basic class has a type tag associated with it.

An important special case of non-basic classes are

Array Classes. Array classes are the subset of non-basic classes that inherit from an `ARRAY` class. Objects can inherit only from a single array class, allowing the array to be allocated as an extensible block of memory embedded inside the rest of the object. This avoids the overhead of having to dereference an additional pointer to the array. Arbitrary array may of course occur as attributes inside an object.

[3] There are some intricacies involved in using explicit dispatching, but the above discussion gives a general idea.

The underlying principle governing class hierarchy structure, then, is that two classes can only be in the same hierarchy if their objects can be merged efficiently into as single memory layout, thus allowing for efficient access code to be generated.

This illustrates a tradeoff between language elegance/programming convenience and code efficiency. Even though we lose the elegance of unifying all classes under one ancestor class, we gain in language simplicity and code efficiency. There is no loss in the power of abstraction because a programmer can still define a non-basic class which may contain an integer or a double-precision floating-point:

```
class INT_OR_DOUBLE is
    i:INT;
    d:DOUBLE;
    ...
end;
```

1.4 Sather Compilation Strategy

We now describe three features of the Sather compilation strategy. The first two are based on intuitive assumptions which will have to be verified later in the study. The third feature is an example of space-for-time tradeoff which was decided in favor of time.

1.4.1 C as Intermediate Language

One of the main goals in the design of the Sather compiler was to make the compiler easily portable to new architectures.[4] At the same time, it seems that C has become useful as an intermediate language as evidenced by the AT&T compiler for C++ and the Eiffel compiler, both of which produce C code. By using C as the "intermediate" language, and since C compilers are available for most architectures, the Sather compiler and programs can be made inherently portable. We also hope to take advantage of the fact that most modern C compilers produce very efficient code. We thus save ourselves the trouble of reinventing the wheel in optimizing compiler technology, and hope to benefit immediatetely from new improvements in this area, which is developing concurrently with processor achitectures.

1.4.2 Dispatch Cache

As discussed above, Sather provides an explicit dispatch which incurs much extra execution costs. How can we lower the costs of dispatching? The belief is that even if a name (say `x`) may refer to different types of objects during the course of program execution, most of the time, `x` will refer to one type of object. Suppose we have:

```
x:$ANIMAL;
x.eat;
```

The hypothesis is that if `x` currently refers to a `REPTILE` object, the next object it refers to will most likely also be in the `REPTILE` class (one could call this principle *class locality*). Following the general principle of making the common case fast, a one-word cache for the target address of the call `x.eat` is allocated. Suppose `x` initially refers to a `REPTILE` object. The first call will result in a miss, and hence the dispatching mechanism is invoked to locate the `eat` routine for `REPTILE`. This result is stored in the cache, along with the type tag for `REPTILE`. If the call is executed again with `x` referring to a `REPTILE` object, the correct `eat` routine is immediately known without invoking the dispatch mechanism. The overhead cost of a hit is then a simple comparison of the type tag against the tag from the previous call. The space overhead is only 2 words per static procedure call.

[4] Sather was originally developed on a Sun4 (SPARC) platform.

1.4.3 Code Duplication

To take full advantage of the Sather features of separate class hierarchies and explicit dispatch, the compiler has to generate 'specialized' code for each instance of an inherited procedure. Consider the following code fragment:

```
class ANIMAL is
   age is <Calculation> end;
   eat is if (age < 30) .... end;
end;

class REPTILE is
   age is <Reptile age calculation> end;
   ANIMAL;
end;
```

Here, `REPTILE` inherits the `eat` routine from `ANIMAL`. Instead of a single shared copy of `eat`, two copies are generated. If there were only one copy of the code, dispatching code would be required to find out the type of object that calls `eat` at runtime, and depending on the type, decide which `age` routine is the correct one to invoke. Furthermore, the class attributes used inside `age` can be at different memory offsets depending on which subclass is used, meaning that attribute accesses would have to be dispatched as well. This would overwhelm the efficiency gained from having explicit dispatch and separate class hierarchies.

Of course, the larger amount of generated code results in higher memory requirements. More memory and disk space is needed to store the class-specific code. It was decided that these were no real issues given today's availability of primary and secondary storage. Potentially more serious would be a performance penalty due to higher cache miss rates. Here again, the principle of class locality can be invoked to argue that the same instance of a procedure will tend to be reused, thus resulting in effective use of the instruction cache. This issue is addressed by the cache performance studies reported later.

1.5 Questions addressed in this study

Having discussed the language design and compilation strategies underlying Sather, we now wish to support the design decisions with data from and analyses of actual Sather programs and their execution on existing hardware.

One source of information is performance data obtained from actually running or simulating Sather programs, including opcode frequencies, cache miss rates, and the like. These measurements can answer several important question.

- Is Sather overall efficient enough to be a viable alternative to, say, C?
- Does Sather code place unusual demands on a standard instruction set architecture, i.e., is there room for improvement at the architectural level (despite Sather being designed for that not to be necessary)?
- Do the CPU and memory usage patterns reveal any weak points in the language design and/or implementation?

Section 2 describes the techniques used in obtaining the required performance data, and section 3 presents the results.

In section 4 we then turn to each of the major language features discussed above and evaluate their impact on performance through a detailed analysis of the performance data obtained and the code produced at the C and assembler levels. This includes conclusions about how Sather code generation could be (And actually was) improved as a result of the findings reported here.

Section 5 discusses remaining issues, such as how the software approach to object-oriented efficiency compares to SOAR's hardware-oriented approach, and what other questions would be worth investigating (but have not been in this study due to time constraints).

Conclusions and final thoughts can be found in section 6.

2 Methodology

One assumption in having Sather compiler generate C code was that the object-orientedness of Sather programs does not require code which is not efficiently expressed in C. Since the state-of-the-art in efficient C implementations is represented by RISC architectures paired with RISC compiler technology, we decided to use a popular RISC platform as our testbed for a comparative study of C and Sather code and performance. Also, since RISC machines are characterized by their lack of special support for language-specific features (object-oriented or otherwise), we expect the results to generalize to future innovations in instruction set architecture developed for general computing.

This section describes the methods used in this study.

2.1 Benchmarks

2.1.1 Microbenchmarks

The bulk of the data comes from a set of small programs designed to highlight one (or a small number of) individual language features at a time. These *microbenchmarks* exist in several versions, including one in pure C, and several ones written in Sather following different programming styles.

Of course microbenchmarks allow only limited conclusions about overall performance of real-world programs. They are, however, useful for a number of reasons. Even if the goal is to optimize the language overall this can be achieved by checking on individual features one-by-one, making sure that none drops far behind. Second, since the programs have a well-defined scope, direct comparisons between C and Sather versions are possible, including inspection at the instruction level. Finally, small programs allow extensive simulations of cache performance, something which isn't always feasible with bigger programs.

The following microbenchmark programs were used:

daxpy The well-known inner loop from the LINPACK benchmark that performs double precision add and multiply on vectors. The program does 100 passes over two vectors of 1000 elements each. Daxpy is intended to measure tight floating point loop performance and array access. Versions include the standard C version, and a direct translation to Sather. As a result of this evaluation the loop-code generated by Sather was substantially optimized. We evaluate both the original and the improved Sather program.

bubble A bubble sort program sorting an array of integers. The program randomly initializes and then sorts a list of 10 elements 10 times and is an example for more general kinds of iterative algorithms that operate on integer arrays.

perm A recursive algorithm that generates all the possible permutations of an array of integers. The program permutes an array of 6 elements 100 times, generating 1237 recursive procedure calls each time.

lst Exercises a stack data structure implemented with self-extending arrays. Creates a list and pushes 5000 integer values onto it.

int_set An implementation of integer sets using extensible hash tables. Various sets are created and operated on using set insertion, union, intersection, and difference.

The first three programs were coded specifically for benchmark purposes, while the last two are extracted from the Sather library. **lst** and **int_set** are typical examples of efficient, general-purpose data structures forming the backbone of most Sather applications. In an object-oriented programming environment they can be reused in a variety of context (due to type parametrization), and by comparing them against hand-coded, specialized C versions we are investigating whether Sather programmers pay an undue performance prize for the programming flexibility gained.

2.1.2 Macrobenchmarks

To complement the data obtained from the microbenchmarks, we also ran one *macrobenchmark* intended to exercise all language features in a more realistic way. As our benchmark program we took the Sather compiler **cs** itself, applied to a very small 'hello world' type program (the Sather compiler is the biggest piece of Sather code to date).

Since there is no Sather compiler written directly in C, we had no way of setting up a direct comparison between C and Sather for **cs** in absolute terms. However, we can usefully compare relative performance characterizations, such as the relative opcode distribution, against that of C programs carrying out comparable tasks. Hence, as reference points for the Sather data in this case we used the instruction set usage data for C programs published in Hennessy & Patterson [4].

We also used the **cs** macrobenchmark to compare different versions of Sather code generation against each other.

2.2 Measurements

As our benchmarking platform we chose a MIPS R2000 based machine, running the RISC/os 4.52 operating system. The main reason for this choice was the availability of profiling tools (*pixie*(1), see below). Also, the MIPS instruction set is closer to DLX than, e.g., the SPARC architecture, making the comparison of our data with those from [4] more straightforward.

2.2.1 Execution timing

For a first assessment of overall relative performance of the C and Sather microbenchmarks, program runs were timed using the *time*(1) available command in UNIX. Except for startup overhead and a small number of memory allocations, the benchmark programs contained no system calls. We therefore compared user CPU time only.

It should be noted that we had to increase the number of top level iterations by factors of 100 or 1000 to obtain enough significant digits from *time*. The run times reported in this context are therefore not comparable to the ones estimated by *pixie*.

All C code, hand-generated or Sather-generated, were compiled with the same maximum optimization level possible[5]

2.2.2 Instruction set usage

Program binaries (compiled from hand-coded or Sather-generated C code) were submitted to *pixie*, a tool that inserts execution profiling code into MIPS loadable object files. During trial runs the binary then generates profiling data that is collected for later analysis. Specifically, we were interested in

- Total number of cycles (i.e., execution time modulo memory system performance, interrupts, and other influences external to the CPU).
- Total number of instructions executed.
- Absolute opcode distribution (i.e., a breakdown of the total number of instructions executed).
- Relative opcode distribution (percentages of total number of instructions executed).
- Instruction concentration (what number of static instructions accounts for a certain percentage of instructions dynamically executed).

[5]On the MIPS, this was not the best optimization level actually available, because that would have precluded separate compilation.

2.2.3 Cache performance

Pixified binaries can also generate address traces. We used the trace data to evaluate cache performance using the *dinero* cache simulator [1]. In particular, we collected data for

- Instruction, data, and combined cache hit rates.
- Memory traffic.

The goal of the cache performance study was to determine the sensitivity of the benchmark programs to varying cache sizes. We therefore varied instruction and data cache size between 1k and 64k bytes, while maintaining a fixed, fairly standard cache architecture, with a block size of 64 bytes, 1-way (direct mapped) associativity, write-back policy, and separate instructions and data.

2.2.4 Data collection

The process of data generation and collection was automated using makefiles. For easy visualization of the results *awk* scripts extracted and formatted the relevant data for graphing.

3 Benchmark Results

3.1 Code size

The Sather compiler and the Sather libraries in their current version have not been optimized yet to minimize executable size, i.e., a lot of unused library code is linked with the main program.

Thus, when comparing Sather programs to their C counterparts, Sather programs yield executables that are between three and eight times as big (both text and total size, see Table 1). This, in turn causes longer delays on program startup and uses more virtual memory.

When comparing the count of (static) instructions actually executed, however, Sather programs turn out to use only about 30% to 60% more instructions. The Sather overhead percentage decreases with the overall size of the program, indicating that Sather programs incur a certain fixed overhead on startup (Table 2).

3.2 Execution time

Table 3 compares overall execution times of C and Sather programs.

Sather programs are very close to their C coded rivals, from slightly better to up to 32% slower. These preliminary (and not very accurate) timing results were meant as a plausibility check for later measurements, which are all based on simulation. The results here are a good enough match to the *pixie* results reported below to lead us to believe that there was no major inaccuracy in those later measurements (such as that pixie does not take varying memory latencies into account).

3.3 Absolute opcode frequencies

We now turn to the absolute opcodes frequencies (including total number of instructions and cycles) determined for the microbenchmarks. Since each of the benchmarks exhibits one or more interesting phenomena we will discuss each in some detail.

3.3.1 DAXPY

The graph (Fig. 1(a)) shows opcode counts for three versions of the **daxpy** benchmark. The first on is the C version (**daxpy/c**), the second one is the Sather program as compiled by the original compiler (**daxpy/sa.old**), and the third one is the same Sather program after some changes to the compiler that improved loop optimization by the C compiler (**daxpy/sa**). Before the change, the C compiler would generate code to recompute the index into the array anew on every iteration, accounting for the singular peek in SHIFT instructions (cf. section 4.2).

Considering now only the C and the improved Sather version, the major difference found is that the C compiler unrolls the DAXPY loop only in the hand-coded C version. As a result, the C version has only 25% of the ADDs (index increments) and branches. The C code is also optimized to implement the loop with a decreasing index, thus eliminating the need for a SET instruction immediately preceding each branch. On the other hand, the C version also incurs more NOPs, which are part of the reason that the substantial advantage in instruction count does not show up in the number of cycles actually used.

As expected from the nature of the benchmark, the number of LOADs, STOREs and floating point operations (FLOPs) in all three versions are identical.

3.3.2 Bubble

The distributions graphed in Fig. 2(a) are from the C version (**bubble/c**), a natural, 'object-oriented' Sather coding that makes use of object attributes (**bubble/sa.oo**), and a Sather version that is optimized in two ways (**bubble/sa**). The first optimization was to make the random number

	C version		Sather version		Ratio	
Program	text	total	text	total	text	total
daxpy	8192	13088	65536	93424	8.00	7.14
bubble	16384	24896	65536	98128	4.00	3.94
perm	16384	24864	65536	97888	4.00	3.94
lst	8192	16384	65536	97504	8.00	5.95
int_set	20480	29088	65536	98784	3.20	3.40

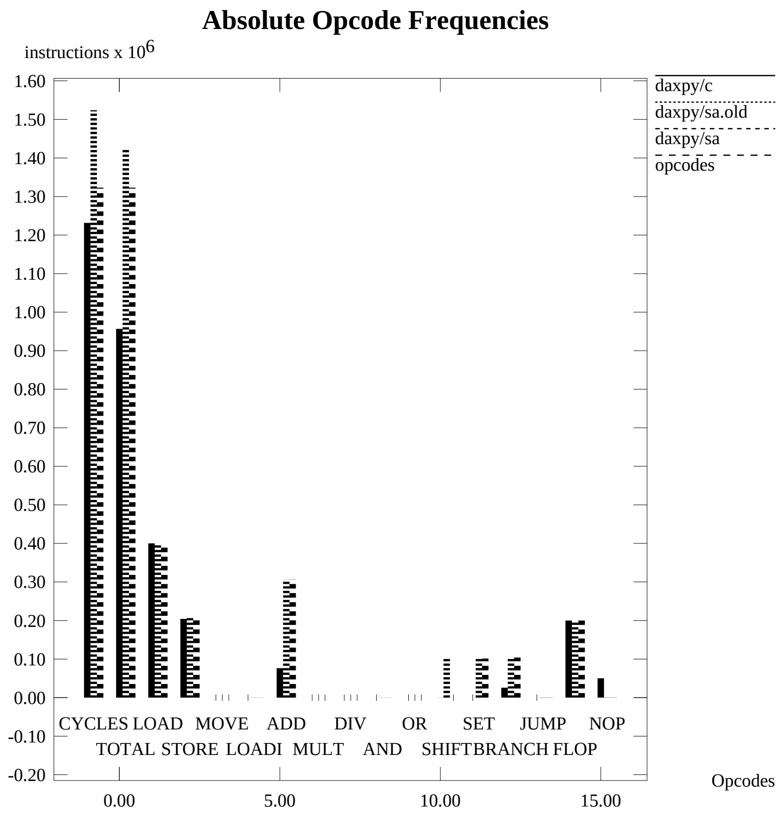
Table 1: Executable sizes in bytes for corresponding C and Sather programs.
Symbol table and other non-runtime overhead are not included (from *size*(1) output).

Program	C version	Sather version	Ratio
daxpy	354	574	1.62
bubble	496	713	1.44
perm	470	637	1.35
lst	498	708	1.42
int_set	880	1171	1.33

Table 2: Number of static instructions executed in corresponding C and Sather programs.

Program	C version	Sather version	Ratio
daxpy	5.31	5.33	1.00
bubble	7.25	7.94–9.60	1.09–1.32
perm	2.22	2.11	0.95
lst	1.20–1.28	1.28–1.36	1.00–1.13
int_set	3.72	3.65–4.37	0.98–1.17

Table 3: User CPU times consumed by benchmark programs.
Times are in second, as determined by the the *time*(1) command. Ranges indicate the minimum and maximum values obtained from different version of the same program.

a.


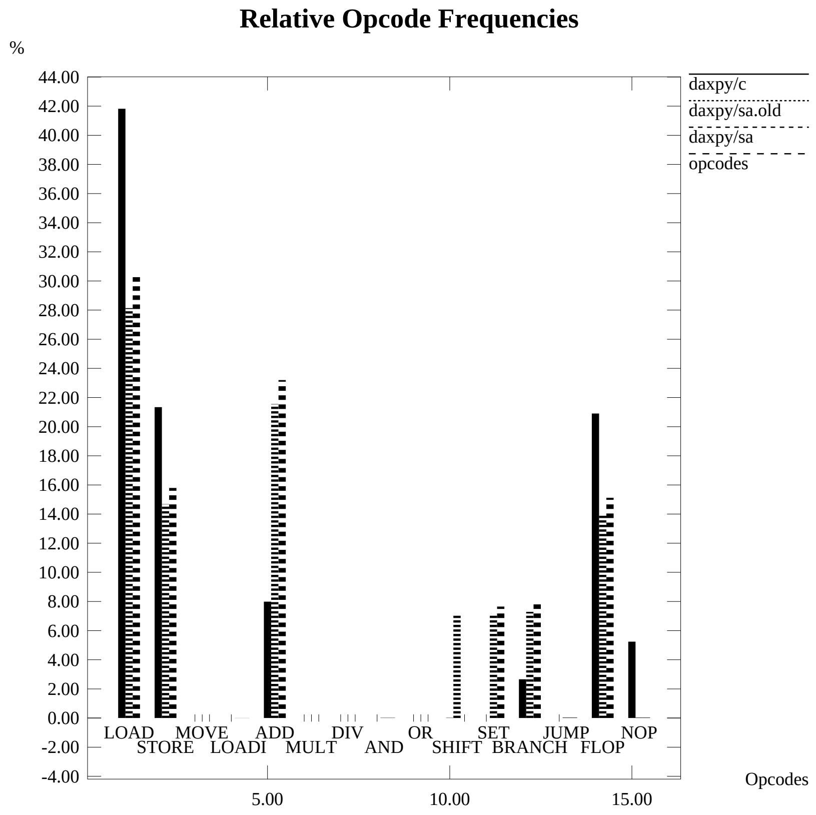
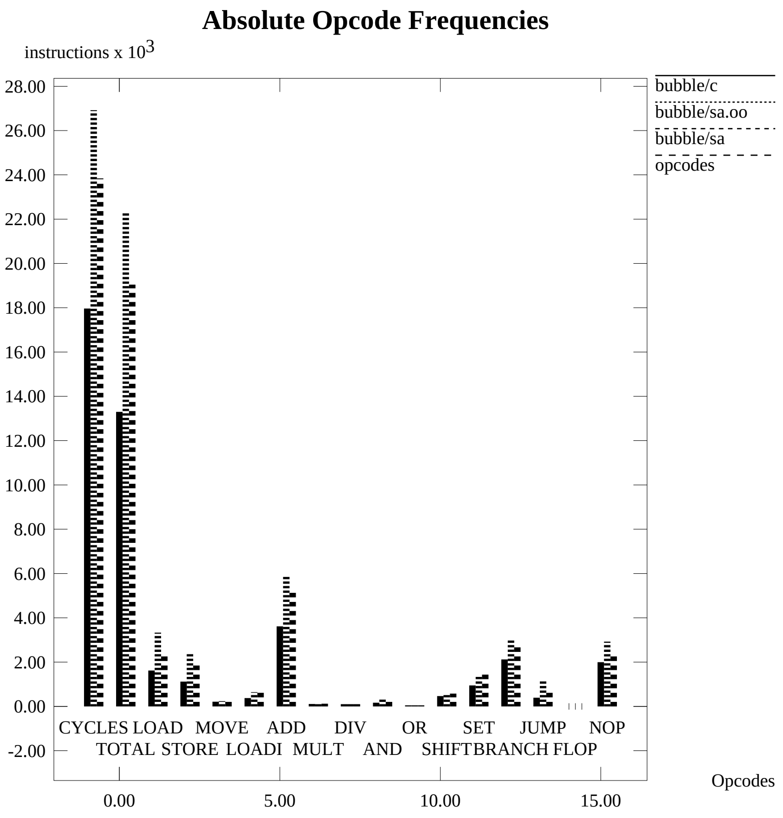
b.


Figure 1: Absolute (a) and relative (b) opcode frequencies for the **bubble** benchmark.

a.


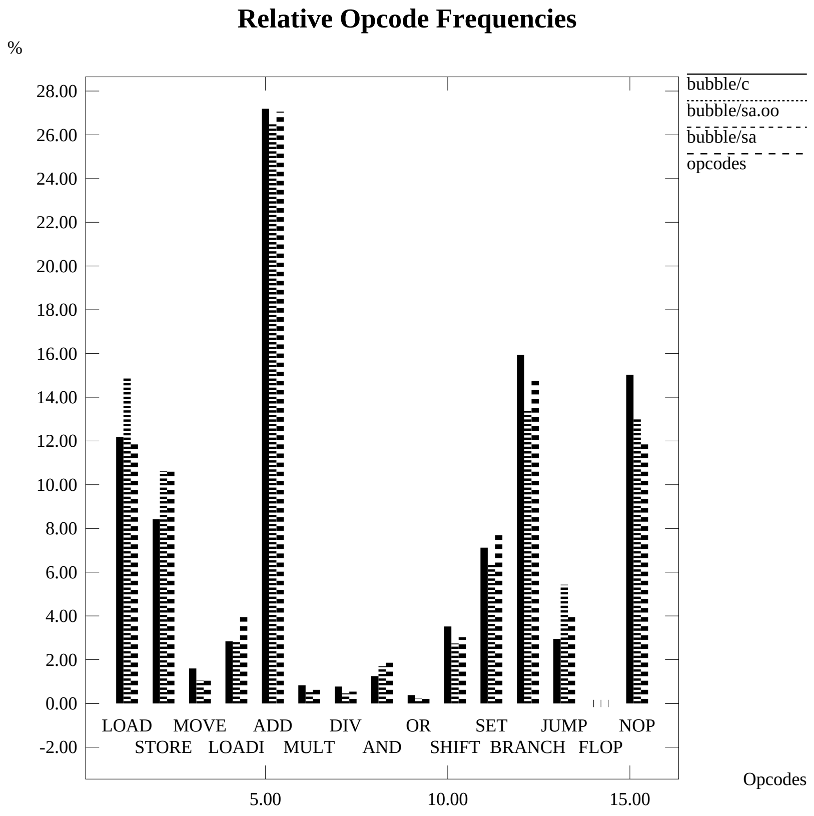
b.


Figure 2: Absolute (a) and relative (b) opcode frequencies for the **bubble** benchmark.

generator used for initialization part of the same class as the sorting routine, avoiding access to a separate class. The second optimization involved using local procedure variables to 'cache' the object attributes (**bubble/sa**), similar to what the C version does. This also allowed reusing the same object for each iteration of the test, avoiding object creation overhead. Because of the many qualitative differences between the unoptimized Sather version and the C version we consider this somewhat of an unfair comparison.

Replacing class attributes with local variables results in a substantial reduction in the number of memory accesses (LOADs, STOREs and ADDs used in address computation). The reason for the higher number of LOADs and STOREs in the optimized Sather version compared to C was harder to find. A substantial part of the runtime is spent initializing the array to be sorted, using a function of no arguments returning a random number. In Sather this function turns into one of one parameter, namely the *self* object, causing a word to be pushed onto the stack for each call. It would be a relatively straightforward optimization for the Sather compiler to omit the *self* argument if the callee is known not to use it. In this case, however, that wouldn't help, because the Sather version of rand() does make use of *self*, namely to access the random generator state that is part the *object*. In C this state is kept in a global variable, but the object-oriented philosophy demands that all data be attached to objects. The resulting argument passing is fairly expensive on modern architectures (memory access) and can be a noticeable overhead, as the example shows.

Control flow at an abstract level is identical in all three versions. However, the number of branches and jumps shows that the C compiler was able to optimize control flow at the machine level, in the following order of optimization quality: C, optimized Sather, standard Sather.

3.3.3 Perm

Figure 3 compares a C (**perm/c**) and a Sather (**perm/sa**) version of the array permutation benchmark. In the C program, the array is accessed through a pointer in a structure, whereas in Sather, the 'structure' is an object that inherits from the `ARRAY` class, causing the array elements to be embedded in the object itself.[6] The **perm** benchmark shows that this language feature gives an substantial performance advantage because many array accesses require only one pointer dereference instead of two, showing up as decrease in the number of LOADs. Also, some offset computation is avoided (less ADDs and shifts).

perm was also chosen as a benchmark because of its high number of procedure calls. It shows that Sather programs which can make use of static dispatching have are as efficient in this regard as C programs.

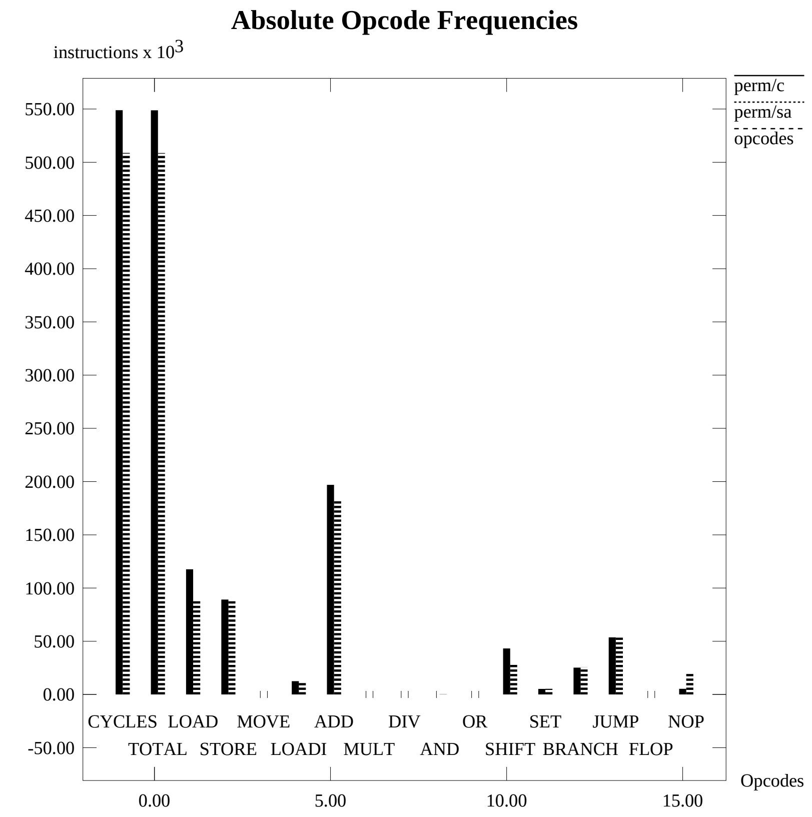
3.3.4 Lst

The impact of several access methods for arrays is further illustrated in the next **lst** benchmark (Fig. 4). The four versions are an 'object-oriented' C version accessing the array through a pointer in a structure (**lst/c.oo**), a simplified C version using global variables and pointers to reference arrays (**lst/c**), a Sather version using array subclassing (**lst/sa.oo**), and a Sather version using objects with array attributes (**lst/sa**).

As expected, the simple C version is the fastest, although the code would not be very useful in practice since it does not allow multiple list instances. The natural Sather version is almost as efficient as the object-oriented C version, while the Sather version that relies on array pointers (rather than subclasses) is the clear loser. Generally, the C versions seem to benefit from better control-flow optimization.

[6] Of course the same memory layout can be achieved in C, but only through 'dirty tricks'. The Sather compiler uses the same dirty tricks but makes them available through an easily understood abstract language feature – array class inheritance.

a.



b.

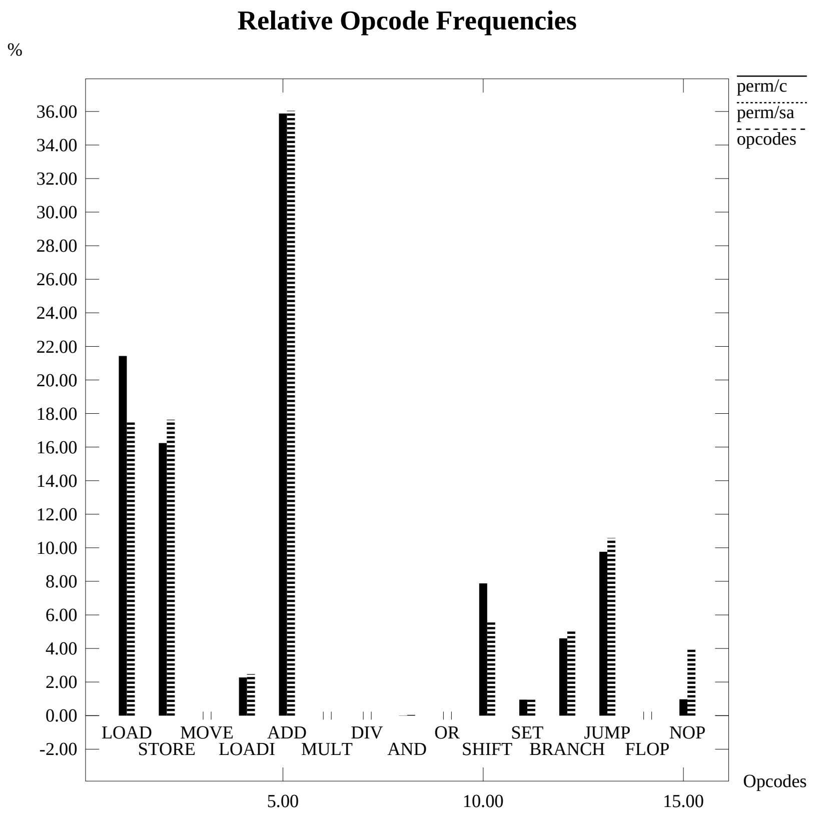
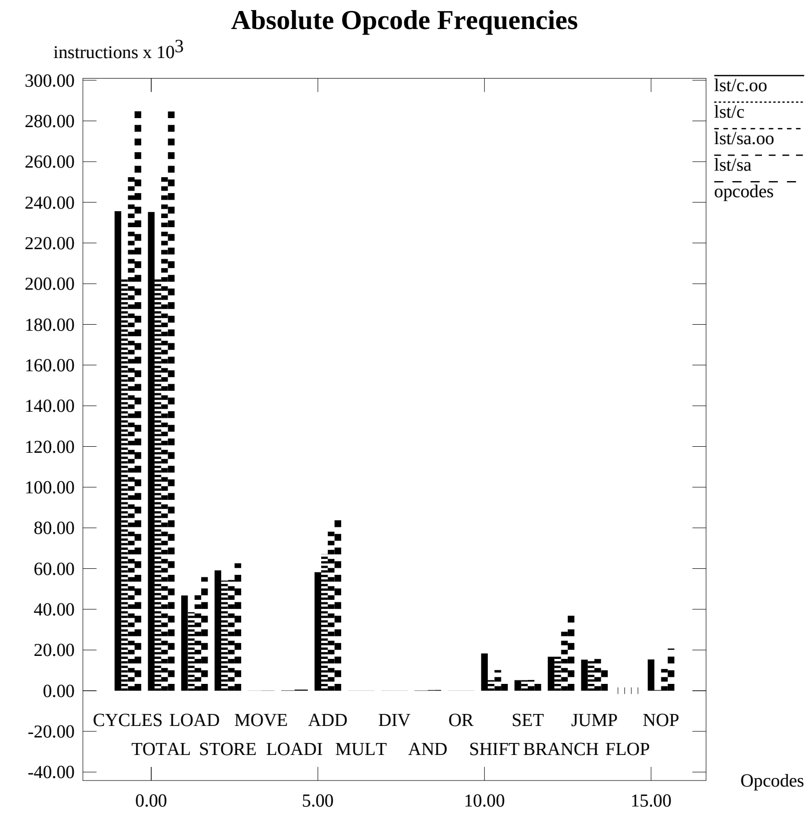
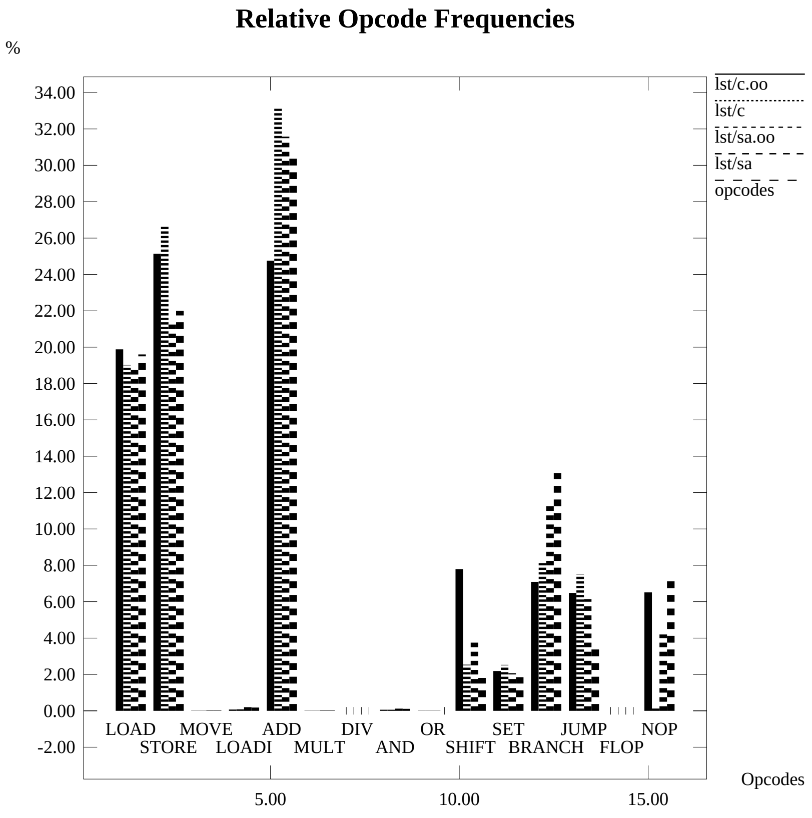


Figure 3: Absolute (a) and relative (b) opcode frequencies for the **perm** benchmark.



a.



b.

Figure 4: Absolute (a) and relative (b) opcode frequencies for the **lst** benchmark.

3.3.5 Int_set

The last microbenchmark returns to the question of how object-oriented code structuring affects code efficiency. The set operations tested in **int_set** benchmark were originally coded in Sather using *cursor classes*. Cursors are an object-oriented abstraction of iterations over sets. This original Sather version (**int_set/sa.oo**) is compared to a C version (**int_set/c**) and an optimized Sather version (**int_set/sa**) which both replace the cursor abstraction with inline code for the iterations.

While the C and optimized Sather version perform almost identically, the object-oriented Sather version pays a clear price in additional memory accesses (LOADs, STOREs), pointer arithmetic (ADD), branches and jumps. Overall, use of the cursor abstraction costs a 30% increase in the number of cycles (according to pixie; the timings reported earlier indicate only a 17% increase).

It should be noted that something like the cursor abstraction is unlikely to be found in C, where it is harder to write general-purpose code that can operate on a variety of data structures (one of the fundamental ideas of object-oriented programming).

3.4 Relative opcode frequencies

While absolute opcode frequencies give an idea of the overall runtime of programs (and which features of a program account for it), relative opcode distributions can give an indication of what part of the instruction set is most heavily used by a certain type of application. In particular, we wanted to know whether Sather programs would generate an instruction mix that had any marked peculiarities compared to what can be found in average C code.

3.4.1 Microbenchmarks

The comparisons at the microbenchmark level in section 3.3 have already shown that there are only minor difference in individual opcode distributions. This impression is confirmed by directly comparing the opcode frequencies plotted as percentages of total (Figs. 1(b)–5(b)).

3.4.2 Macrobenchmark

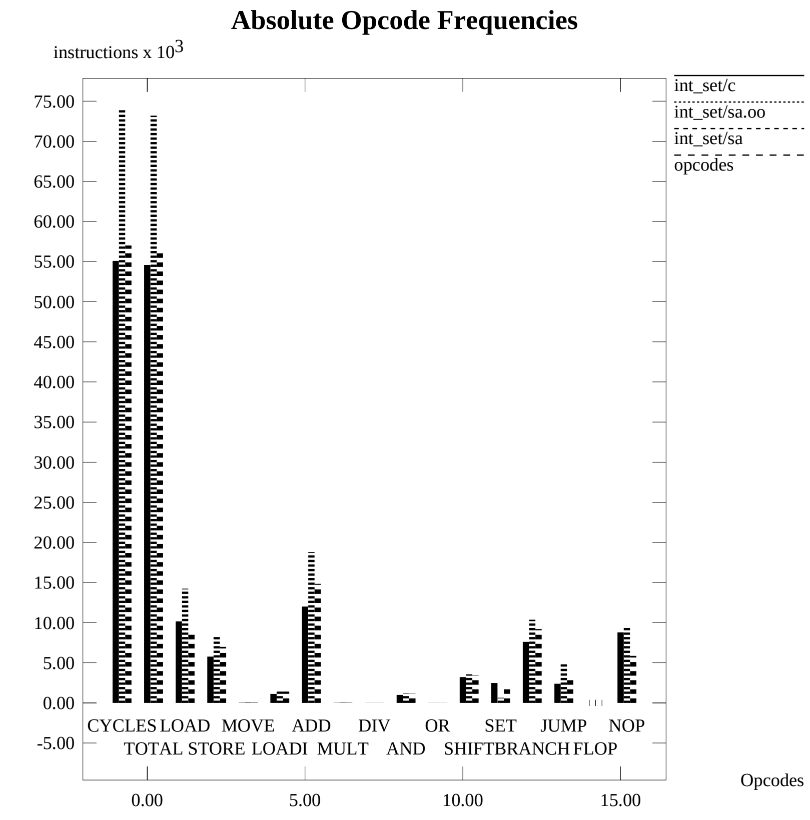
Part of the reason microbenchmark profiles for C and Sather are so similar is that each microbenchmark performs a highly specific task that largely determines the types of instructions used. It is therefore important to complement these data with distributions from large-scale applications with a more realistic instruction mix.

Figure 6 compares the relative opcode distrubution obtained in a run of the Sather compiler (**cs/sa**) to that of two C applications, the GNU C compiler (**h+p/GCC**) and the TeX text formatter (**h+p/TeX**). The C data is taken from the instruction mixes reported for DLX in [4] (DLX is a generic, fictitious RISC instruction set architecture that is close enough to the MIPS for our purposes).

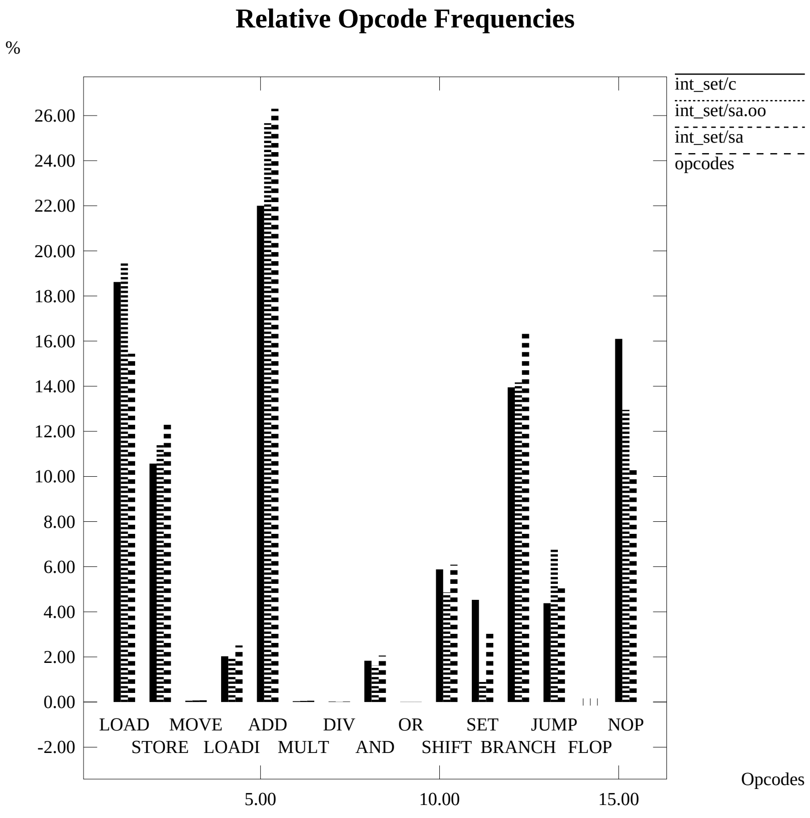
For most opcode groups, Sather frequencies are within 1% of one of the two C programs (LOADI, ADD, AND, SHIFT, SET, 1% being the accuracy with which this data is given). For LOADs, STOREs, and branches, Sather falls between the values of GCC and TeX.

The only abnormality in Sather is the high number of jumps (7.44%). Closer examination of the indidual opcodes involved reveals that this figure is entirely due to procedure calls (JAL 2.05%, JALR 0.17%, JR (including procedure returns) 2.46%). PC-relative jumps are only slightly more frequent than in GCC (2.75%). We attribute this pattern to the way **cs** was coded, as a collection of a large number of small procedures calling each other. The nature of the code is also evident in the average number of instructions per basic block: 5.

Note that NOPs are not accounted for in Patterson & Hennessy's data. Nevertheless the percentage in Sather (17.9%) seems unusually high. Again, we think the fragmented character of the code can account for this. A certain number of the NOPs (14%) can be attributed to the jumps

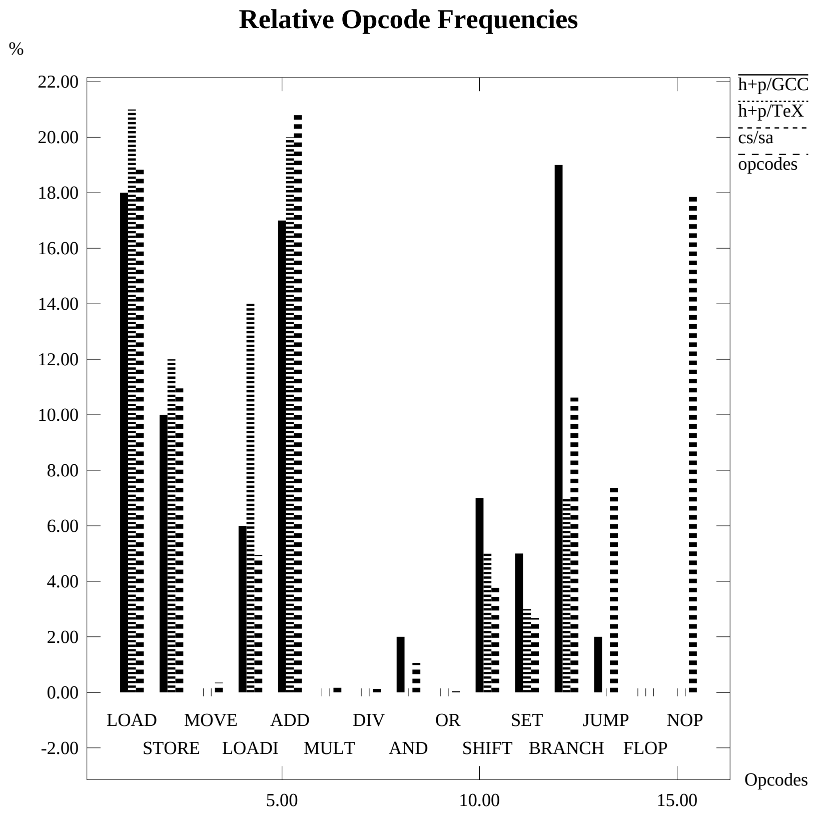


a.

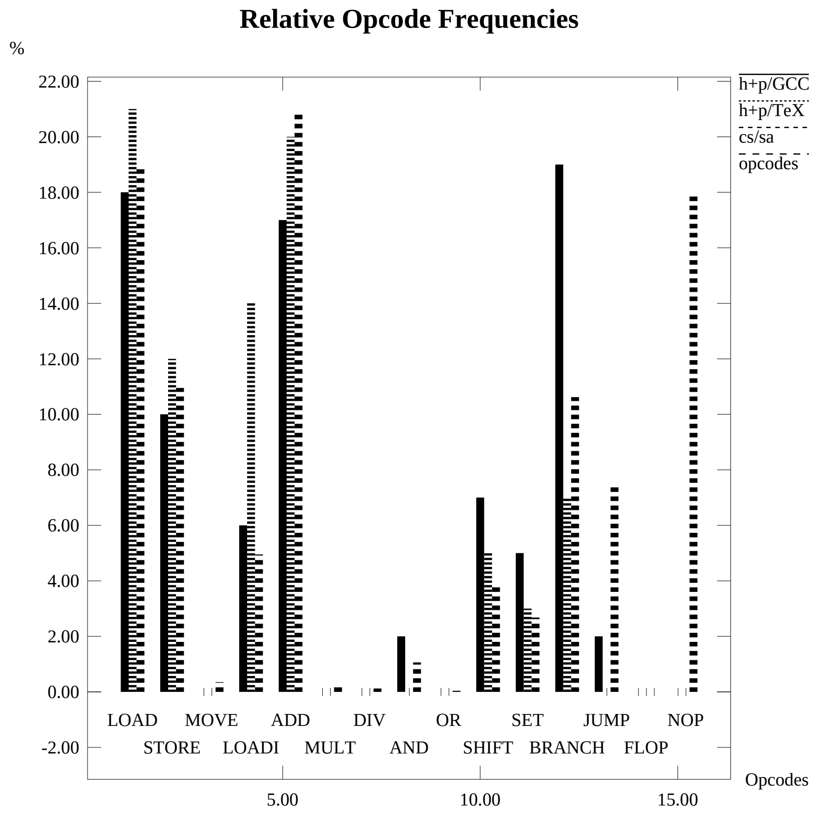


b.

Figure 5: Absolute (a) and relative (b) opcode frequencies for the **int_set** benchmark.



a.



b.

Figure 6: Instruction mixes for the Sather compiler and two C applications reported in [4].

Program	Instructions	C version	Sather version
daxpy	32	83.6%	99.7%
bubble	64	76.2%	61.8%
perm	32	74.1%	75.4%
lst	64	90.7%	97.0%
int_set	128	74.1%	69.5%

Table 4: Instruction concentration in benchmark programs.
Shown are percentages of dynamic instructions accounted for by the most heavily used (static) instructions (number given in the first column).

themselves. More importantly, the small average size of basic blocks seems to prevent effective instruction reordering to circumvent NOPs in branch delay slots (52.7% of all branches) and load NOPs inserted safeguard against read-after-write hazards (43.2% of all loads).

3.5 Instruction Concentration

As noted earlier in section 3.1, Sather programs tend to be more voluminous, although the actual number of instructions used is only moderately higher (cf. Table 2). This fact can be characterized more precisely in terms of *instruction concentration*, i.e., the number of static instructions responsible for a certain percentage of instructions dynamically used.

Table 4 gives a rough picture by listing the percentages obtained by a fixed number of instructions. Although the Sather versions have higher overall number of instruction in each case (Table 2), C has lower instruction concentration in three of the benchmarks. Of these, **perm** and **lst** can be attributed to be more efficient array memory layout. The anomalous result for **daxpy** is explained by the loop unrolling, which the compiler performs only on the C version.

3.6 Cache performance

As noted before, cache effectiveness was simulated as a function of cache size, using a 64-byte block, direct-mapped, write-back cache in all cases.

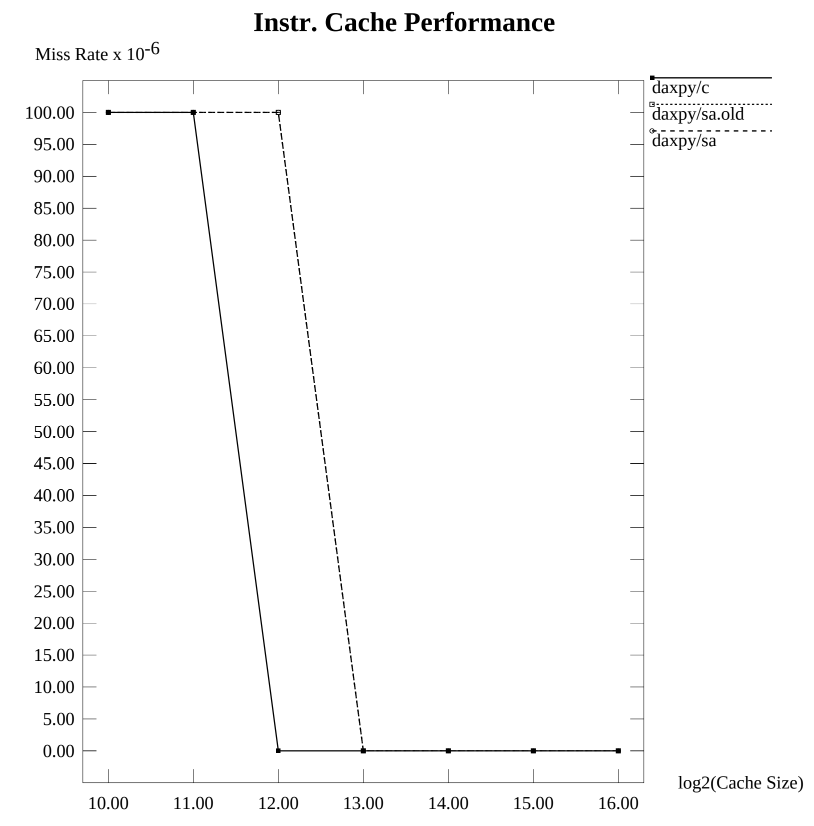
3.6.1 Instruction cache

Instruction concentration by itself has little relevance for instruction cache evaluation, although it is intuitively related to both spatial and temporal locality. One reason is that the number of instruction involved is small enough to fit into a realistically sized cache in all cases, but only under the assumption that these instruction are actually adjacent in address space. Also, effects due to cache architecture (such as cache conflicts) have to be simulated.

Direct simulation of instruction cache performance on the five microbenchmarks gives a very consistent picture. Figures 7(a)–11(a) show that the instruction miss rates in Sather drop along curves that are parallel to those for C programs, lagging behind by varying amounts. In most cases, doubling the cache size is enough to achieve the same hit rate as in the corresponding C program. The final hit rates (for large caches) are the same.

3.6.2 Data cache

The picture for data cache performance is more complex that for the instruction cache (Figs. 7(b)–11(b)).

a.


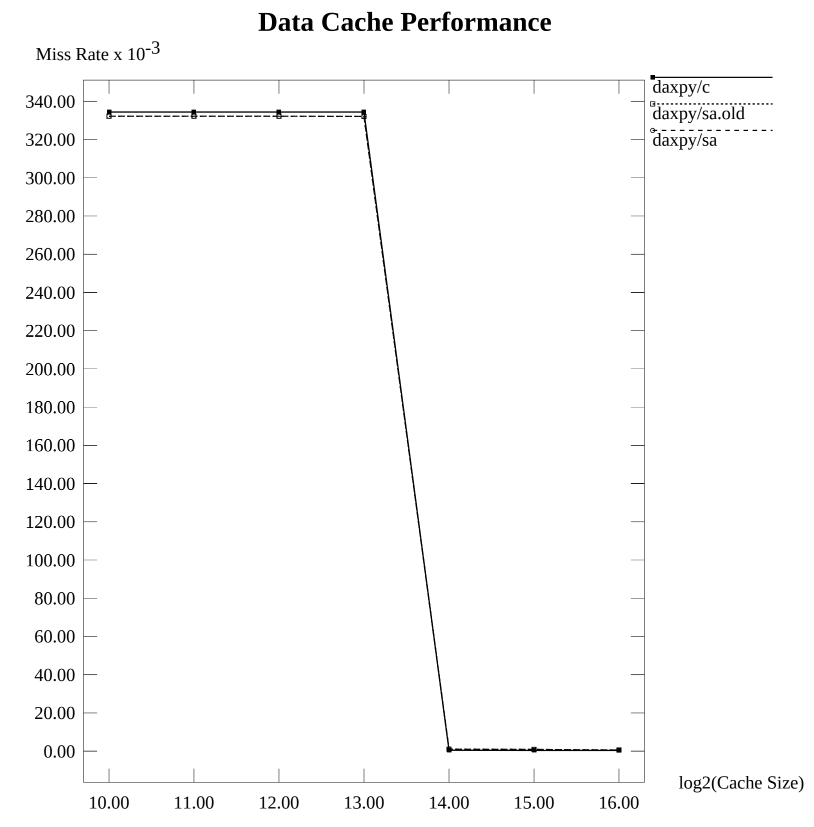
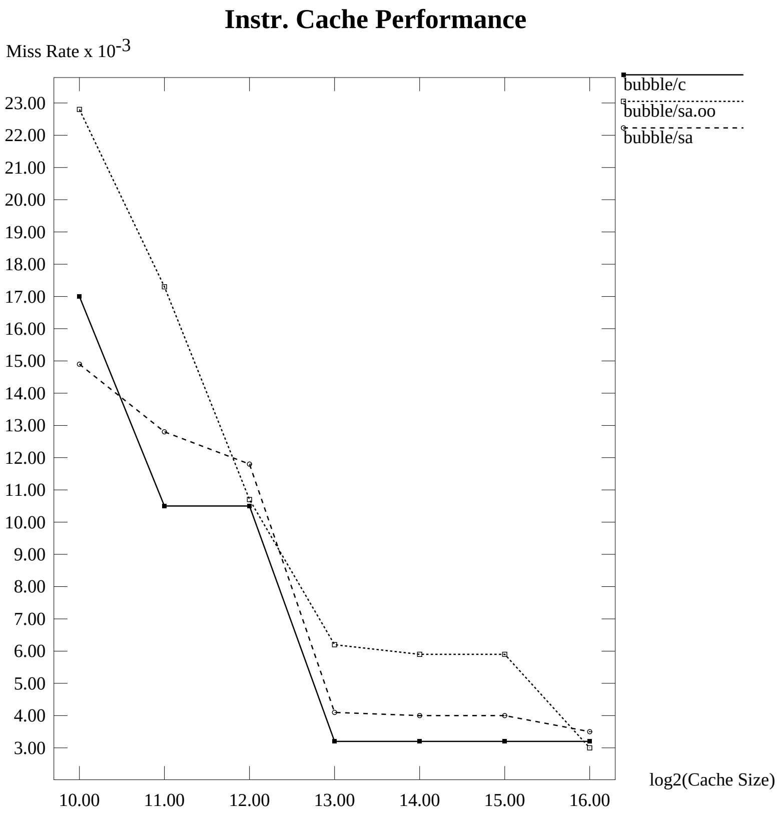
b.


Figure 7: Instruction (a) and data (b) cache miss rates for the **daxpy** benchmark. Cache size varies from 1k to 64k each.



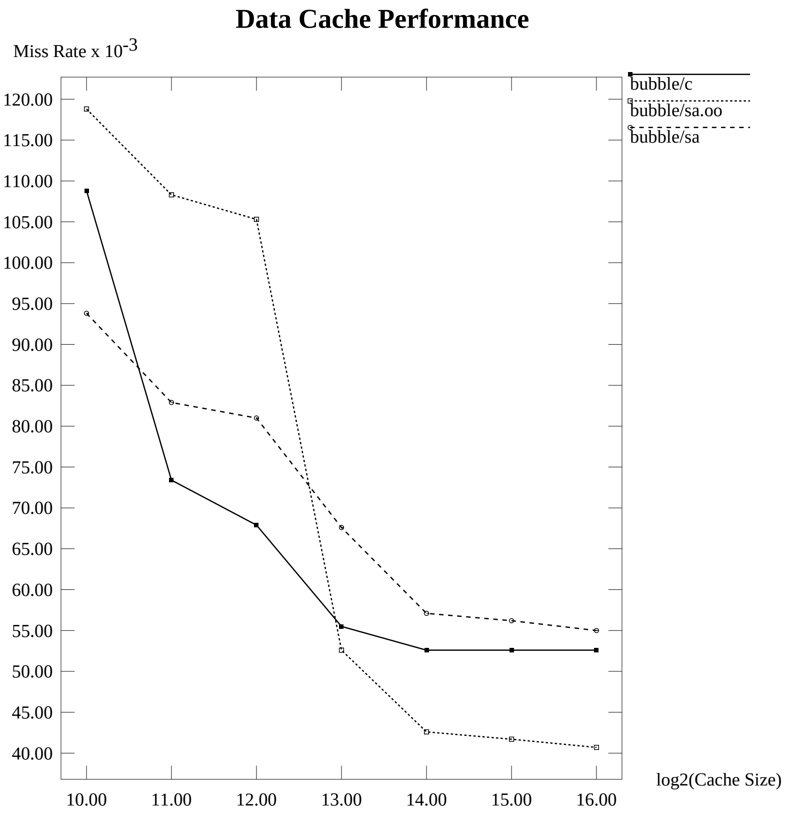
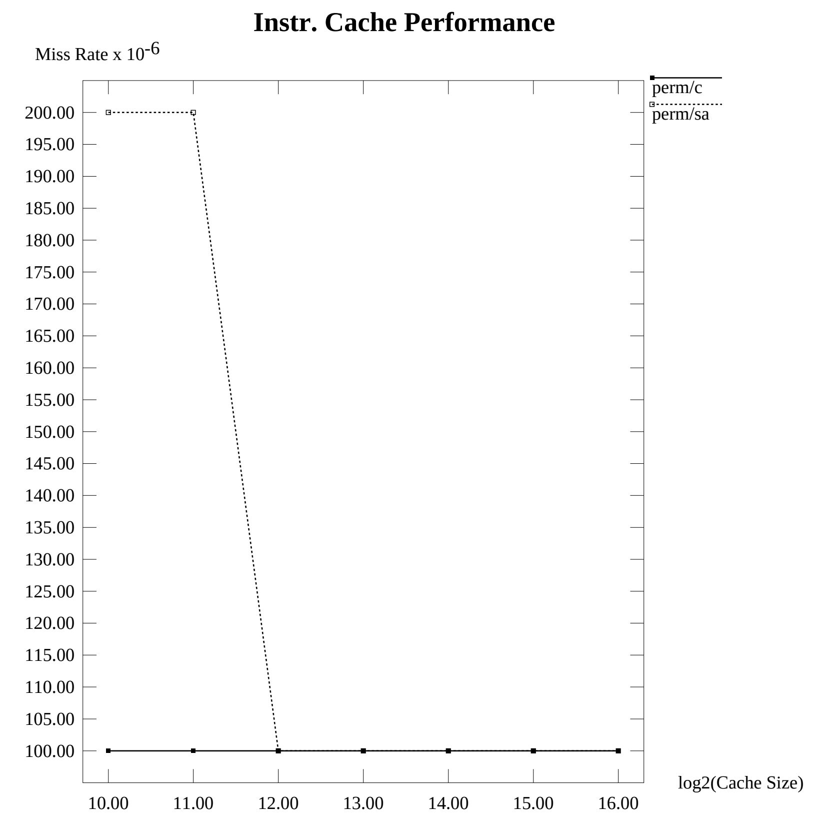
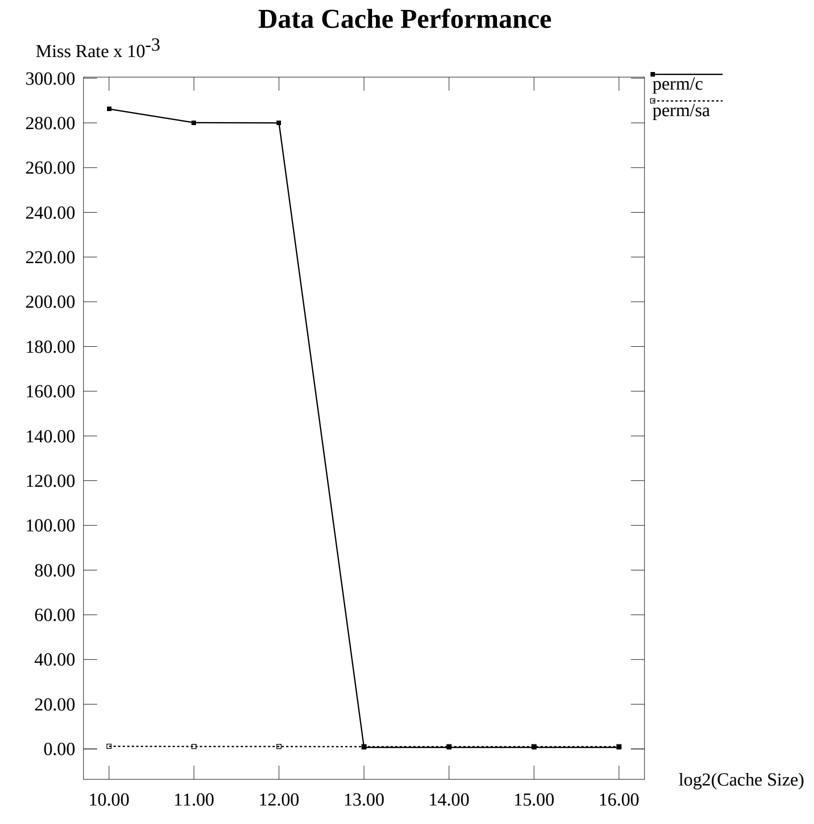


Figure 8: Instruction (a) and data (b) cache miss rates for the **bubble** benchmark. Cache size varies from 1k to 64k each.



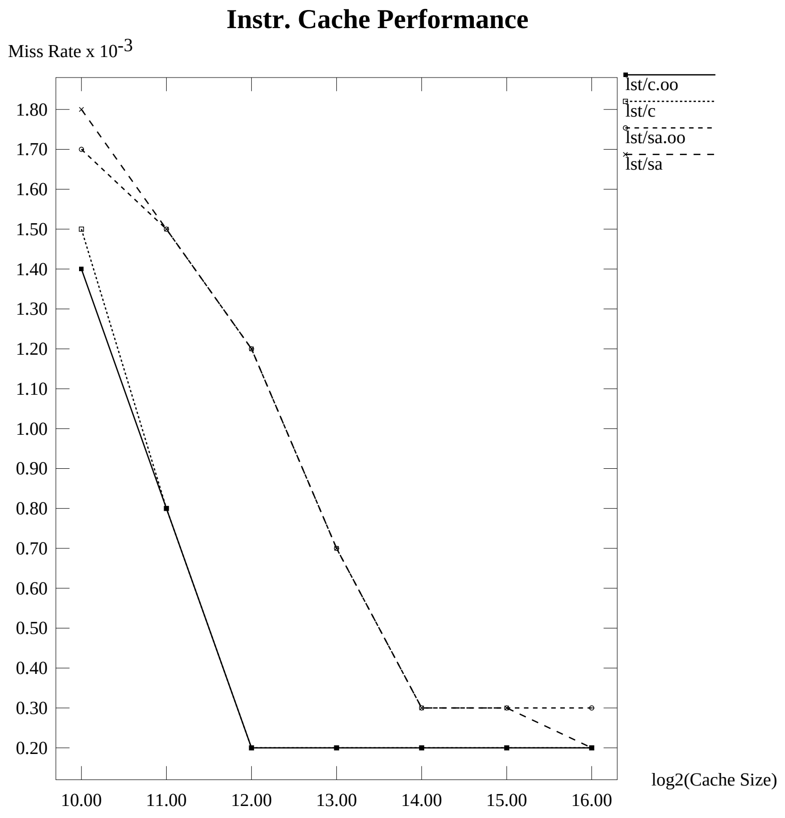
a.



b.

Figure 9: Instruction (a) and data (b) cache miss rates for the **perm** benchmark. Cache size varies from 1k to 64k each.

a.



b.

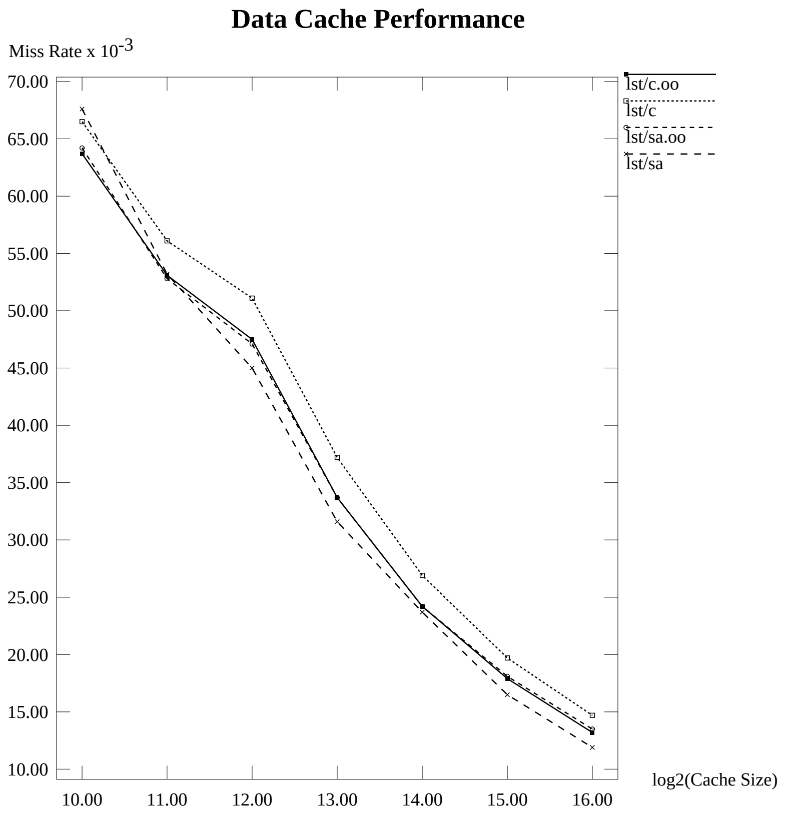
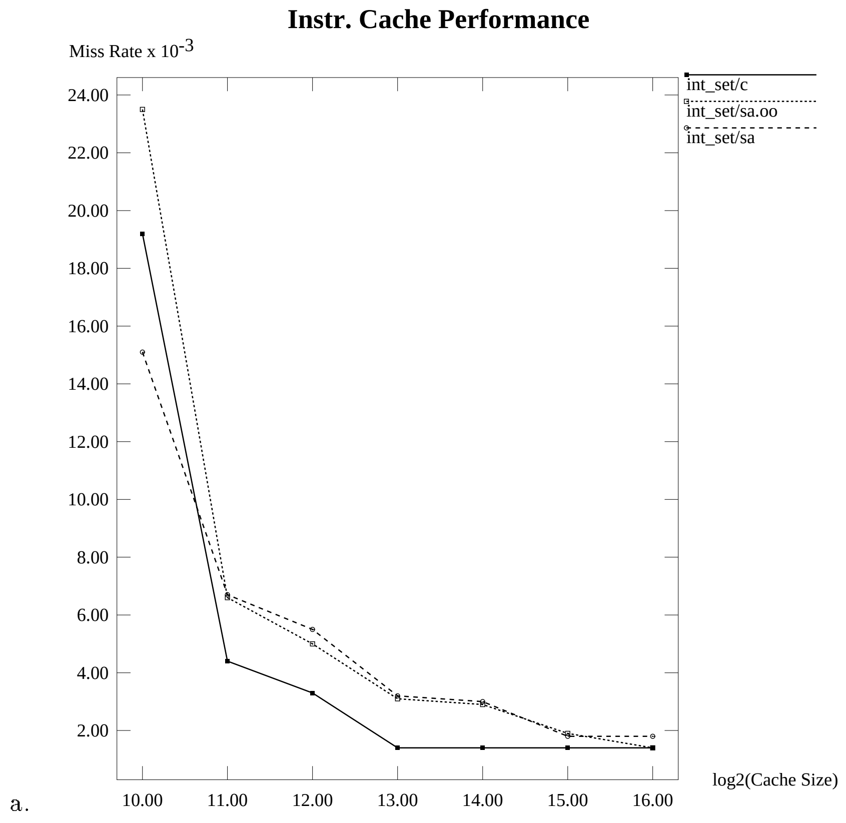


Figure 10: Instruction (a) and data (b) cache miss rates for the **lst** benchmark. Cache size varies from 1k to 64k each.



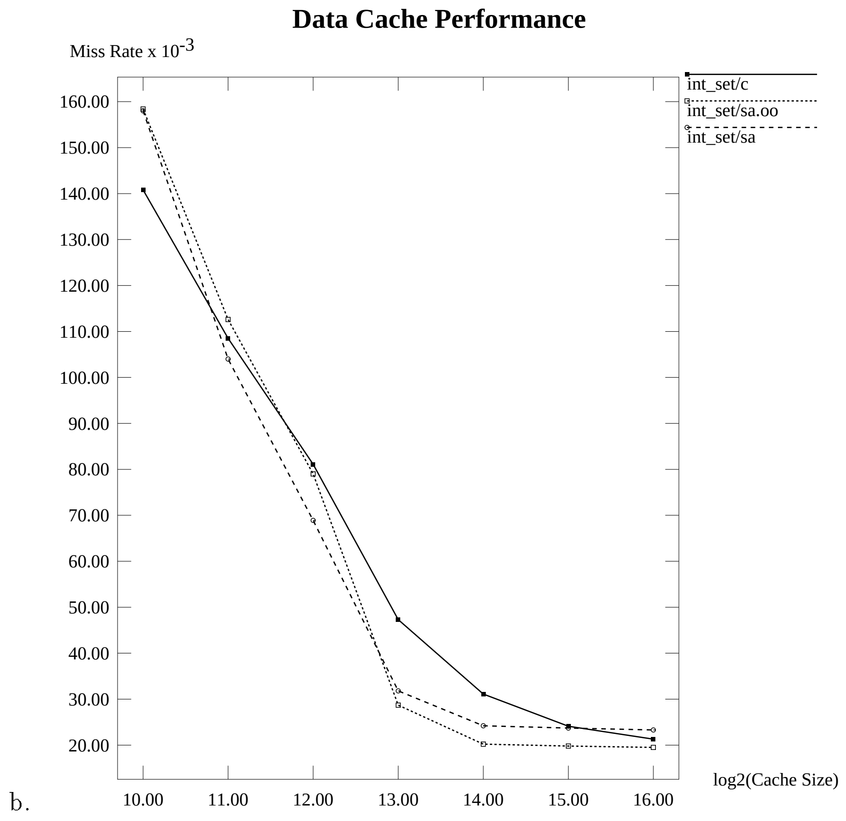


Figure 11: Instruction (a) and data (b) cache miss rates for the **int_set** benchmark. Cache size varies from 1k to 64k each.

For several benchmarks the working set size is entirely determined by the algorithm, and using similar data structures, almost precisely the same miss rates result. This is the case for **daxpy** (Fig. 7(b)), **lst** (Fig. 10(b)), and **int_set** (Fig. 11(b)).

bubble shows the natural object-oriented Sather version slightly lagging behind the C version. The hand-optimized Sather version is worst initially, but achieves the lowest miss rate eventually, possibly because a cache conflict encountered in the other two version is avoided (we found no good explanation for this behavior).

In the **perm** benchmark, the Sather version (using an array subclass) achieves a perfect hit rate even with small caches, whereas the C version starts out with a high miss rate, until the cache is big enough to hold both the array and the structure containing the array pointer. This shows nicely that the efficient memory layout chose for Sather array subclasses can afford a substantial advantage also with regard to memory system performance.

3.6.3 Memory traffic

Memory traffic in all benchmarks closely follows data cache miss rate. Sather versions typically show a certain lag with respect to C versions, but follow the same profile. An example is **int_set** (Fig. 12(a)). In the case of **perm** the performance advantage for Sather on small data caches carries over to memory traffic ((Fig. 12(b)).

3.7 Overall performance evaluation

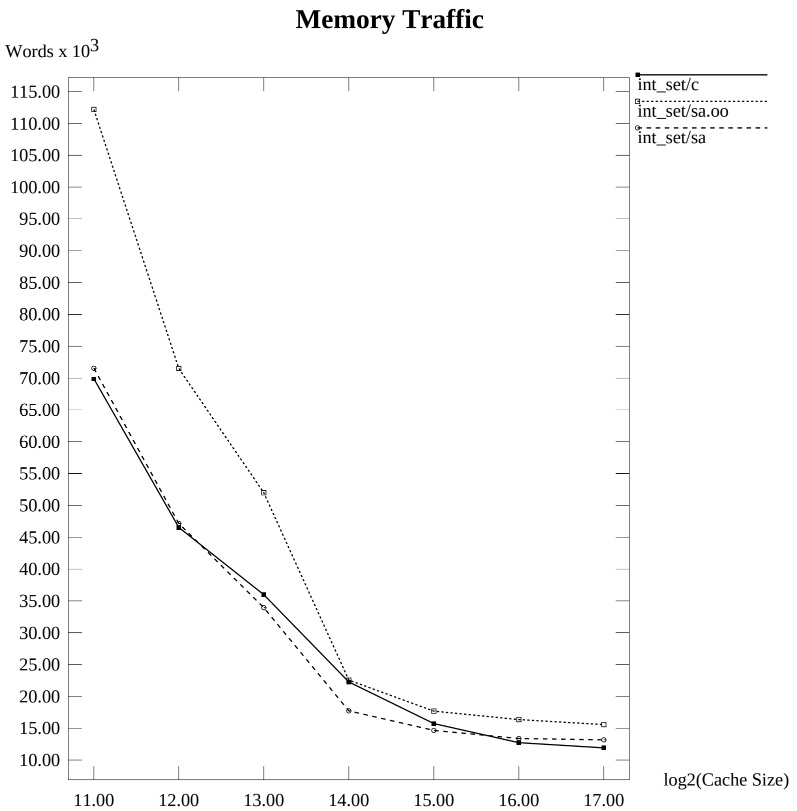
Overall, the microbenchmarks yield a very consistent picture of Sather performance relative to C. In terms of both CPU resources (cycles) and memory resources (cache usage and memory traffic) Sather programs tend to pay a slight penalty for the conveniences of 'object-orientedness'. However, the difference are either insignificant or small enough to be tolerable, especially given the pace of current developments in both CPU performance and cost of memory.

In some cases (objects with a single array that can be efficiently 'inlined') Sather data structures result in code that is both slightly faster and more cache-efficient than standard C data structures for the same task.

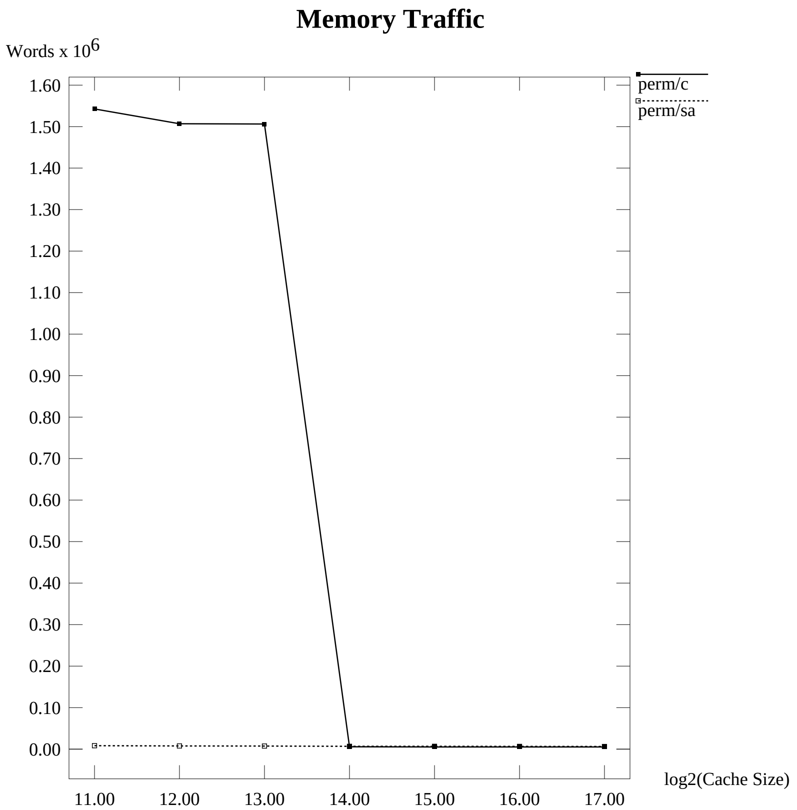
As mentioned previously, Sather language design anticipated and accepted the need for relatively large amounts of memory. Our cache studies show that larger caches may be even more important than larger memories to avoid a memory bottleneck.

The most important conclusion for us at this point is that the comparison between Sather and C yielded only quantitative differences (if any) and no qualitative ones. Instruction mixes and cache effectiveness profiles show characteristic shapes that are strikingly similar or identical.

We take this to be an indication that reduced object-oriented languages like Sather are a good match for standard architectures (which were originally developed for procedural languages like C), and that they are certainly suitable for contemporary RISC architectures.



a.



b.

Figure 12: Memory traffic (in words) as a function of cache size for the **int_set** (a) and the **perm** benchmarks.

Cache size varies from 2k to 128k (instruction + data).

4 Evaluation of Sather Implementation and Design

In this section we take a closer look at some of the details of Sather implementation and design as they affect performance. In particular, we evaluate the results of our benchmark studies and code analyses with respect to the implementation strategies outlined in earlier (in section 1.4).

Some familiarity with Sather [3] or object-oriented languages in general may be helpful.

4.1 Portability

The first version of the Sather compiler was written in Sather and bootstrapped on a SPARC platform. In the introduction, we mentioned that portability was one of the reasons for using C as an intermediate language of the Sather compiler. Since we chose MIPS as the testbed platform for this study, we were forced to evaluate the portability goal as part of the procedure. Since we had Sather-generated C code for the compiler on the SPARC, we simply had to move the C code over to MIPS and made a small number of changes by hand to adjust to the different compilation environment. In effect, the Sather compiler on the SPARC served as a cross-compiler for itself on MIPS. The entire porting procedure took about one night, the major adjustments being

- The Sather environment needs a certain fixed directory structure, where basic classes and command files are stored. This directory hierarchy had to be recreated on the target machine.
- Certain makefiles had to be changed to provide certain machine-dependent flags to the local C compiler (e.g., to increase the default the symbol table size to cope with the high demands imposed by the **cs** program).
- Some platform-specific C library functions were not available and the corresponding Sather procedures had to be eliminated.[7] None of these function (e.g., for timing) were crucial for the functioning of the compiler.

From this experience and a separate port of the compiler to Sequent,[8] we feel confident that the decision to use C as an intermediate language was the right one, as it actually delivers on the promise of portability. One caveat here is that all architectures used so far with Sather have very similar in several crucial respects (such as memory alignment properties and the UNIX environment). It is likely that other less standard platforms will present more of a challenge.

4.2 Loops and Array Access

From an initial study of the **daxpy** benchmark, we found a source of inefficiency in the Sather-generated C code (though this does not apply to the SPARC C compiler).

First we show the loop structure of Sather-generated C code which is as follows:

```
while (1) {
   <preliminary code>
   if <cond> break;
   <statements>
}
```

The preliminary code consists of code to perform any dispatching or runtime checks before the test expression is evaluation. This portion of the code may be empty. Given the above loop structure, the MIPS C compiler generates the following code sequence:

[7] Sather programs can call C code via the `C` class.

[8] The Sather compiler was ported to Sequent by Franco Mazzanti.

```
LS:      # Start of loop
    <preliminary code>
    Branch to LE if <cond> holds
    <statements>
    Branch to LS
LE:      # End of loop
```

Thus the MIPS C compiler was not smart enough to generate the following more efficient sequence of loop code that eliminates one branch instruction.

```
    <preliminary code>
    Branch to LE if <cond> does not hold
LS:      # Start of loop
    <statements>
    <preliminary code>
    Branch to LE if <cond> does not hold
LE:      # End of loop
```

Hence, we performed an experiment by manually altering the Sather-generated C code. The transformations include:

1. **Loop statement**. We transform the loop statement to the following form:

```
    <preliminary code>
    if <cond> goto L1;
    while (1) {
       <statements>
       <preliminary code>
       if <cond> break;
    }
L1:
```

2. **Array Access**. The original Sather-generated C code that accesses the *i*th element of an array is (after macro expansion):

```
(*((double *)((x__)+( 8 + ((i__) << 3)))))
```

We changed the access to the following form:

```
((double*)x__)[1 + (1 * (i__))]
```

(The hand-transformed code follows the syntax used by the Sather-generated C code.) The performance results can be seen from the following statistics gathered from pixie. In the first hand-altered version (a), we incorporated only the change in the loop statement. In version (b), we incorporated both changes in the loop statement and array access.

Compiler	**Number of cycles**	**Execution Time**(s)
(i) Sather	1726323	0.0691
(ii) Hand-compiled (a)	1523881	0.0610
(iii) Hand-compiled (b)	1323083	0.0529
(iv) C	1231609	0.0493

We note that the pure C code was about 40% faster than the original Sather-generated C code. However, the (pure) C code is only about 7.3% faster the (better) hand-compiled (Sather-generated) code. The manual code-alteration improves the time of Sather-generated C code by about 30.6%. Hence altering the form of the loop-code allows the C compiler to generate better code. The difference between (iii) and (iv) is due to loop-unrolling performed by MIPS C compiler.

After altering the code generation of the Sather compiler to generate the new code sequence for loop-statements, we obtained the following exactly the Sather performance expected from line (iii).

This shows that altering the form of Sather-generated C code can help the C compiler produce better executable code. However, as demonstrated by (ii) and (iii) in the table, the Sather compiler runs the risk of trying too hard in second-guessing the C compiler's optimization strategies. For example, when the array access was

```
(*((double *)((x__)+( 8 + ((i__) << 3)))))
```

we obtained less speedup than when we changed the Sather-generated code for array access to[9]

```
(*((double *)((x__)+( 8 + (8 * (i__))))))
```

Modified compiler	**Number of cycles**	**Execution Time**(s)
(i) Loop + Array access	1323083	0.0529
(ii) Loop	1523881	0.0610

The reason is that the MIPS C compiler was able to perform strength reduction on the induction variable `i_` when `i_` was multiplied by 8. However, when the left-shift operation is used, the MIPS C compiler lost the ability to do strength reduction.

To verify the usefulness of the new loop-structure, we tested the old and new Sather compilers on a simple code fragment (*F1*):

```
for (i = 0; i < 10000000; i++) {
   j += (k + 1);
   k = (j + 3) + k * 3;
}
```

In addition to running pixie, we did a *gprof* profile of the program. The results are as follows:[10]

Compiler	**Number of cycles**	**Profiled Time**(s)
(i) Sather (new)	80004877	3.2400
(ii) Sather (old)	110004880	4.4600
(iii) C	65000422	2.6400

Again, the difference between (i) and (iii) is due to the loop-unrolling performed by the MIPS C compiler. To illustrate that the better performance is due to the loop-unrolling, we re-installed the new Sather compiler on SPARC, and re-run the test-code. The SPARC C compiler does not perform any loop-unrolling with the standard optimization option (**-O**). We obtained the following results from profiling.

Compiler	**Profiled Time** (s)
(i) Sather (new)	3.61
(ii) Sather (old)	3.63
(iii) C	3.67

[9]The C compiler treats `((double*)x_)[1 + (1 * (i_))]` and `(*((double *)((x_)+( 8 + (8 * (i_))))))` as equivalent.

[10]The time reported by pixie is the same as the time given by the profiler

Hence the improvement made with respect to the MIPS C compiler was irrelevant on the SPARC, due to the difference in the C compiler.

We wanted to find out if the Sather compiler can somehow take advantage of the loop-unrolling done by the MIPS C compiler. By modifying the C code, we found that:

- The MIPS C compiler performs loop-unrolling for `while` statements as well.
- The `if + break` statements at the end of each loop prevented the MIPS C compiler from performing loop-unrolling on the Sather-generated C code.

This suggests that an alternative code-template might allow us to take advantage of C compiler's optimization. To verify our hypothesis, we took the code fragment (*F1*) and hand-compiled it using the following code template for the loop statement.

```
     <preliminary code>
     while (!(<cond>)) {
        <statements>
        <preliminary code>
     }
 L1:
```

Not surprisingly, the MIPS C compiler was able to perform loop-unrolling for the resultant C code. Hence, in the future, we may be able to improve the Sather compiler further by using the alternative code template for the loop statement.

4.3 Dispatch Efficiency

As discussed in the introduction, there were several assumptions implicit in the design of dispatching:

1. Most of the time, a programmer knows that a name can refer to exactly one type of object. Hence, a language with explicit dispatch will result in programs that drastically reduce the number of dispatches.
2. There is a certain amount of "constancy" with respect to the type of objects referred to by a name (class locality), so that dispatch caches will be effective in reducing the costs of dispatching mechanism.

In this section, we discuss whether the results found in our experiments support our assumptions. There are two distinct sets of data we wish to collect.

1. **Effectiveness of Explicit Dispatch**. The question we want to ask here is: Given the ability to specify dispatching explicitly in the Sather language, how effectively does a programmer make use of his/her knowledge of the types of objects to reduce the amount of dispatching?
2. **Dispatch Cache Hit/Miss Rate**. How effective do dispatch caches reduce the costs of dispatching?

For the first question, we are asking how many extra dispatches will be incurred. if a programmer does not have the option of specifying dispatch explicitly. To gather such statistics, we modify the Sather compiler to go through an extra pass (just before code generation) to mark all objects (whose types are non-basic) as being dispatched.

The simulation of all-dispatched classes in Sather is made under the following assumptions:

Void Object. No invocation is made on void objects in Sather programs.

Certain valid Sather programs become invalid under the assumption that object accesses or routine calls are dispatched. For example, if we have:

```
x:ANIMAL;
x.create;
```

If the `create` routine in class `ANIMAL` does not access any attribute of the object, the call `x.create` works perfectly well in Sather, even though `x` may not refer to any object. However, in fully-dispatched object-oriented languages, during execution, the type of object referred to by `x` must be known in order to call the correct `create` routine. Since `x` does not refer to any object in this code, the execution will fail.

Code duplication. Code duplication is done to avoid dispatching; this is possible even in object-oriented languages which assume all types to be dispatched.

Suppose we have the following definition:

```
class ANIMAL is
   age:INT;
   eat is if (age < 30) then ... end; end;
end; -- class ANIMAL
```

In the implementation of some object-oriented languages, if the class `REPTILE` inherits from `ANIMAL` and does not redefine `eat`, objects in both classes then share the same code fragment. This means that we do not know the exact offset of the attribute `age` until execution when the type of object that invokes `eat` is identified. This incurs extra dispatching cost. However, if the code for `eat` is duplicated for class `REPTILE`, then the reference to `age` in the duplicated code clearly refers to a `REPTILE`'s `age` and no dispatching is necessary. This technique can also be applied in object-oriented languages with all-dispatched types, and hence we consider this an orthogonal alternative in our measurements.

Separate Basic Class Hierarchies. The disjoint basic class hierarchies were an independent design decision. Hence we maintained this assumption, in the measurements, so the dispatches do not include use of `INT`, `CHAR`, `BOOL`, etc.

To verify the assumption of class locality, the Sather runtime support was modified to count the number of hits and misses each time an attempt is made to dispatch an object. This arrangement allows us to measure how effectively the implementation strategy (dispatch cache) works when explicit dispatching is available. Intuitively, since the programmer specifies dispatching explicitly in the program, we expect that a name `x` refers to objects of varying types during the course of program execution. The question then is: How much variation is there?

Measurements are gathered for two types of dispatches. (a) The first type of dispatch is to determine the correct feature (attribute / routine / shared / constant) of a class to be used. (b) The second type of dispatch is to determine the base size of an array to access the ith element of the array.

The measurements show that the dispatch caches are actually useful in reducing the amount of dispatch computation. (NOTE: The version of Sather compiler used to gather the following statistics is the one that generates improved loop code (as described in section 4.2.)

P0-P3: We execute the Sather compiler with the respective inputs being (i) the compiler, (ii) a simple program that creates two arrays, (iii) a simple test program that does dispatch, and (iv) test program on the class `INT_SET`.

N1-N3: We execute the Sather compiler with the inputs being (ii), (iii) and (iv) in the previous case. In these cases, the programs have already being compiled before, so the Sather compiler will try to avoid generate any new code.

Program	No. of Hits	No. of Dispatches	Hit Rate
(i) *P0*	1326447	1700726	0.78
(ii) *P1*	29710	36517	0.81
(iii) *P2*	31156	38895	0.80
(iv) *P3*	35778	46249	0.77
(v) *N1*	29649	36404	0.81
(vi) *N2*	30787	38290	0.80
(vii) *N3*	34323	44086	0.79

From the above results, and other test runs (executing the compiler with different options), the hit rate stays quite stable at around 80%. Hence we are confident that in the normal case, the dispatch cache will be useful in reducing execution time.

The other set of measurements is to find out how many extra dispatches will be incurred if Sather does not provide explicit dispatch as one of the language features. To measure this, we generate a new version of the compiler which assumes that all types are dispatched. We then re-run this new compiler with the same inputs (as in the previous table) to get the new number of dispatches, and cache hits.

The following table shows the number of dispatches when the new compiler is executed with the same set of inputs

Program	Type (a) Dispatch	Type (b) Dispatch	Total
(i) *P1*	190752	188920	379672
(ii) *P2*	202903	197638	400541
(iii) *P3*	228964	216635	445599
(iv) *N1*	194181	188611	382792
(v) *N2*	204394	196926	401320
(vi) *N3*	225715	214737	440452

The following table shows the reduction in the number of dispatches by having explicit dispatch in the Sather language.

Program	Explicit Dispatch	All Dispatch	% Reduction
(ii) *P1*	36517	379672	939.7
(iii) *P2*	38895	400541	929.8
(iii) *P3*	46249	445599	863.5
(iv) *N1*	36404	382792	951.5
(v) *N2*	38290	401320	948.1
(vi) *N3*	44086	440452	899.1

If we disregard the dispatch done to access base size of array objects, the table is as follows:

Program	Current	Type (a) Dispatch	% Reduction
(ii) *P1*	36517	190752	422.4
(iii) *P2*	38895	202903	421.7
(iii) *P3*	46249	228964	395.1
(iv) *N1*	36404	194181	433.4
(v) *N2*	38290	204394	433.8
(vi) *N3*	44086	225715	412.0

Timing execution time of the compiler on the hello-world program we get:

	Explicit Dispatch	**All Dispatch**
Total No. of Dispatches	35648	373653
Timing Profile (s)	0.638	0.710

The reduction in execution time is 11.3 % as compared to 947.0 % in the number of dispatches. Even though the reduction is not drastic, it is still quite significant.

To take a look at the cost of dispatching, from the compilation of the hello-world program, we get:

	Explicit Dispatch	**All Dispatch**
No. of Misses	6512	8091
No. of Cycles	543257	720899
Avg. Miss Cost (cycles)	83.4	89.1

The above table does not include the cost of testing the object type (when there is a hit) and cost of updating the cache (when there is a miss). There is room for improvement in the dispatching mechanism, though the cache has served well so far. Fig. 13 gives a breakdown of the absolute opcode frequencies, using three different versions of the Sather compiler.

cs/sa.dsp This version of the compiler was generated using a compiler option that ignores dispatching and generates code to do dispatching for all object accesses and routine calls. It was compiled with the C optimization flag on.

cs/sa.unopt This version of the Sather compiler uses explicit dispatching and was compiled without C optimization.

cs/sa This version of the Sather compiler uses explicit dispatching and was compiled with the C optimization flag on.

One point to note is that while the **sa.dsp** uses 11.3% more cycles than **sa**, **sa.unopt** performs even worse, consuming 16% more cycles than **sa**. This illustrates the importance of C optimization in Sather code efficience as C optimization allows us to get more speedup than the explicit dispatch feature in the language.

From the previous table and Fig. 13, we can conclude that there is a definite performance advantage to be gained by having explicit dispatch in the Sather language.

4.4 Code Duplication

In section 1.4.3, we discussed the reasons for duplicating code in the case of inherited procedures. Although the importance of this feature seem quite obvious by itself, we verify the performance impact by generating a version of the **perm** benchmark that simulates the effect of non-duplicated code. Instead of altering the compiler for this purpose, we use the compiler option that turns off explicit dispatch. The **perm** code is modified so that each reference to a feature in the same class is explicitly accessed via `self`. For example, a code fragment:

```
class ANIMAL is
   age:INT;
   eat is if (age < 30) then ... end; end;
end; -- class ANIMAL
```

is modified to become

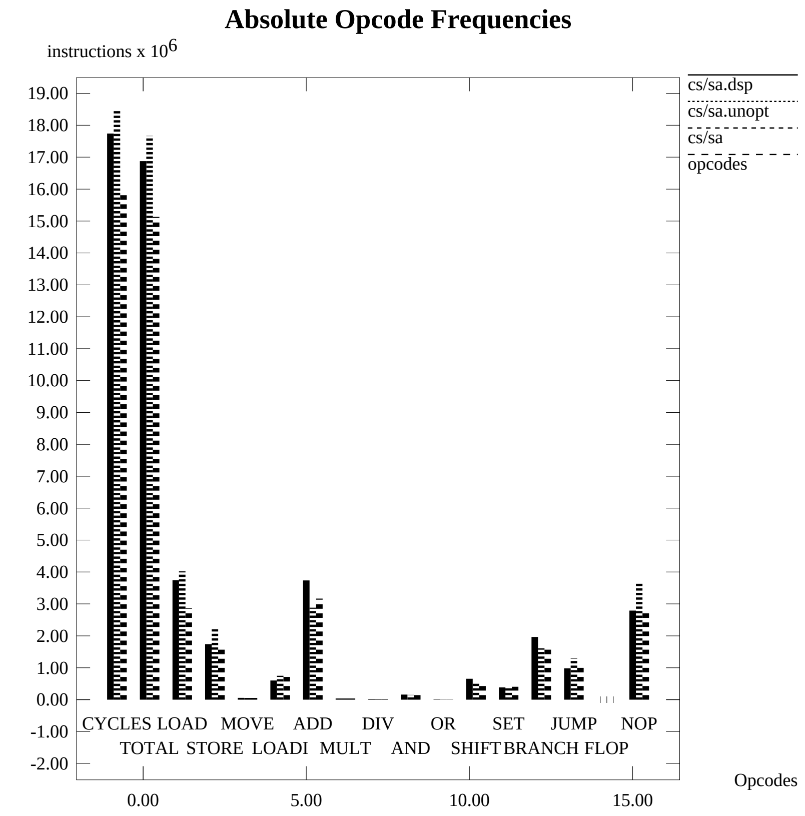


Figure 13: Opcode frequencies for different versions of Sather compiler.

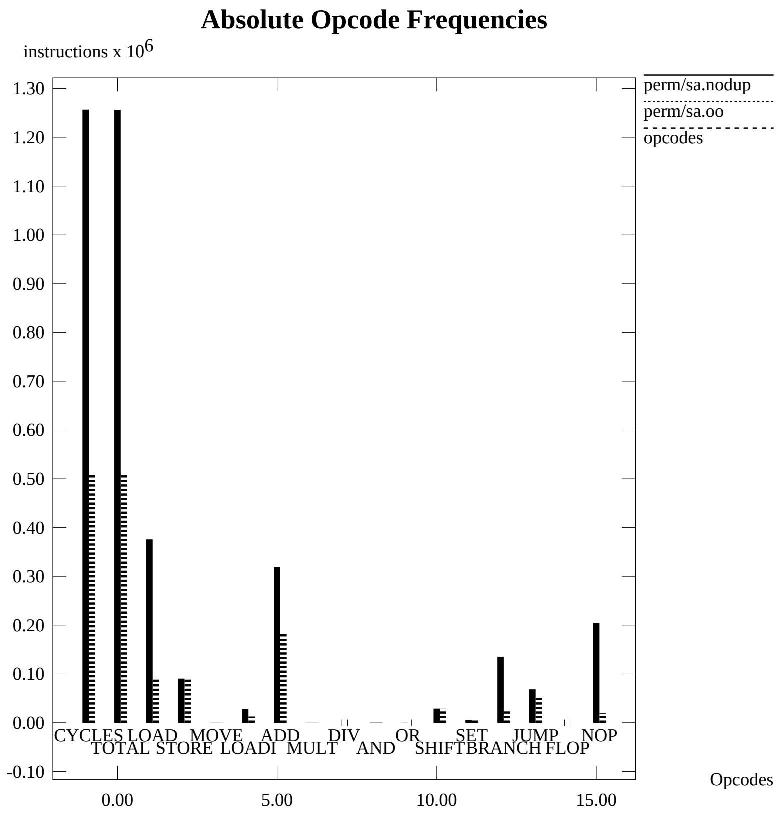


Figure 14: Opcode frequencies for the **perm** benchmark with and without code duplication.

```
class ANIMAL is
   age:INT;
   eat is if (self.age < 30) then ... end; end;
end; -- class ANIMAL
```

Both code fragments are semantically equivalent in the Sather language and implementation. However, if the 'dispatch all' compiler option is set, then the call `self.age` becomes a dispatched call. This is exactly the code that would be generated if there were no code duplication.

The effect of having code duplication can be seen in Figure 14. Pixie profiling shows that eliminating code duplication would have resulted in about a 146% increase in the number of execution cycles. In particular, the number of LOADs and branches is greatly increased by the dispatching code generating for each call and attribute access.

Again this confirms the implementation strategy of trading space for time.

5 Other Issues

In the following, we complement the previous results and discussions with a more detailed comparison to SOAR and a list of other possible evaluation criteria for Sather.

5.1 SOAR vs Sather

We discuss the differences/similarities of SOAR and Sather with respect to three object-oriented features that have traditionally been performance bottlenecks for object-oriented programs.

5.1.1 Arithmetic/Comparison Operations

As mentioned in the introduction, these operations are supported efficiently at the machine level. However, the generality of classes such as `INT` have prevented the object-oriented language compilers from generating code that uses the machine operations directly. SOAR and Sather have different approaches for overcoming this dilemma. We look first at SOAR.

Tagged data. SOAR supports tagged data. To reduce the cost of arithmetic and comparison operations, SOAR checks the tags and performs the operation simultaneously. According to [6], over 90% of the "+" operations perform integer addition. Hence, most of the time, the result is available after 1 cycle. If the data types are wrong, SOAR traps to routines that carry out the appropriate computation for general data types.

Tagged immediate operands. SOAR's immediate operand format was also modified to accommodate tagged data. The high order 4 bits of the 12-bit field becomes the tag bits of the operand, while the lower order 7 bits and the 8th bit of the field form the low order 7 bits of the operand and the sign-extension respectively.

In Sather, the problem is handled by the language design itself. As discussed in section 1.3, Sather introduces separate class hierarchies for basic types, thus eliminating any need to check data types during execution of arithmetic and comparison operations. From experience, the disjoint basic class hierarchies do not impose any significant constraint in programming practice.

5.1.2 Dispatch Mechanisms

Both Sather and SOAR use the idea of caching to reduce the costs of dispatching. However, the implementations differ. In SOAR, the target address of a call is cached in-line in the instruction stream. This requires SOAR to support non-reentrant code. On the other hand, since Sather programs are compiled into C, the cached value is kept in a per-process data area. In addition, Sather language has explicit dispatch, allowing the programmer to avoid dispatching where possible.

5.1.3 Garbage Collection

Both SOAR and Sather provide garbage collection. In all Sather benchmarks (including the macrobenchmark) garbage collection was not required since virtual memory runtime environments usually provide enough memory for short-running applications. Sather provides an off-the-shelf garbage collector for cases where memory recycling is required. In this case the compiler helps the allocation mechanism because it can generate an extra parameter that tells the memory allocator whether the object to be allocated may contain additional pointers to other objects.

SOAR, on the other hand, was designed for a long-running, interactive Smalltalk system which could not tolerate noticeable delays in response due to off-line garbage collection. This prompted development of a high performance generation scavenging garbage collector that is certainly more sophisticated than the one used by Sather (according to [6], it uses only 3% of the CPU time in SOAR).

5.2 Other Possible Evaluation Criteria

There are other criteria by which Sather programs can be evaluated against other object-oriented systems. This is list of loose ends not followed up in this study because of time constraints.

5.2.1 Class Attribute Access

A class in Sather is almost like the `struct` construct in C. Hence, an attribute in a class would correspond to a field in a C `struct`. The Sather compiler treats all Sather objects as consisting of a sequence of bytes. Attribute access is performed by adding a byte offset to object pointer. One question that arises is: Is attribute access in Sather as efficient as field access in C?

From a comparison of the assembly codes generated from Sather and C code fragments, we obtained the following:

```
Sather        :  a1.x := 2;
Sather-C      :  IATT_(a1__,T1_24_OF_x_) = 2;
MIPS Assembly :  li      $14, 2
                 sw      $14, 12($2)

C             :  a1->x = 2;
MIPS Assembly :  li      $14, 2
                 sw      $14, 0($2)
```

This example shows that Sather attribute access is equally efficient to field access (via pointers) in C, at least for the implementation on MIPS.

5.2.2 Higher-dimensional array access

We compare the assembly codes generated from Sather and C programs in the **daxpy** program. In the case of one-dimensional arrays, there appear to be no difference in the efficiency. For example, storing a double-precision floating-point number 1.0 in Sather and C results in the following code fragments:

```
Sather        :  y[i] := 1.0;
Sather-C      :  DATT_(y__, 8 + (8 * (i__))) = 1.0;
MIPS Assembly :  s.d  $f0, 8($3)

C             :  y[i] = 1.0;
MIPS Assembly :  s.d  $f0, 0($3)
```

Two-dimensional arrays in Sather are implemented by storing offsets of each row at the base of the array. A similar idea applies to higher-dimensional arrays. Due to time constraint, we were not able to investigate the efficiency of this implementation strategy, and evaluate other possible implementations.

5.2.3 Virtual Memory/Swap Space Requirements

The size of compiled Sather code tends to be larger than size of compiled C code (cf. section 3.1). The reasons are:

Code Duplication: The same Sather code may be replicated several times, each time with specific attribute values.

Large Runtime Support: There is a large amount of code for runtime support and routines of predefined classes.

In our micro-benchmarks, the extra code is due to the second factor, because class inheritance is not used. In this study, we only study the effect of code sizes on caches. There is another question which we did not have the time to investigate: How would the support for virtual memory (e.g. amount of swap space) affect the performance of Sather programs?

Also, there is as yet no experiment to try out different garbage collection strategies in the Sather environment. Since different garbage collection algorithms work well with different virtual memory strategies, another interesting problem which was not investigated will be to study the best memory support/garbage collection strategy for Sather.

6 Conclusion

The overall picture emerging from the study is very encouraging. The runtime studies have shown that the principal design goal, efficiency relative to procedural languages, was met very well. In cases where Sather programs incur an overhead compared to C it seems small enough to be justified by the considerable advantages object-oriented programming affords in terms of programming convenience and productivity.

Another conclusion is that Sather programs generate the same instruction mix contemporary instruction set architectures were designed to implement efficiently. No particular object-oriented demands on the hardware seem to be left over in a 'reduced object-oriented language' like Sather.

We were able to justify each of the major design and implementation decisions involved in Sather: intermediate C code for portability and compiler optimization, disjoint class hierarchies, explicit dispatch and code duplication for efficient object and procedure access, and call target caching to minimize dispatching overhead.

As a result of the study, we have identified (and partially implemented) several possible improvements to Sather code generation, with several others certainly still to be found. Also, there are aspects of Sather programs worth investigating in more detail for an even better understanding of performance issues. Nevertheless we feel confident saying that Sather design and implementation have more than held up to their goals and now represent a very attractive environment for efficient object-oriented programming.

Acknowledgements

The work reported here originated as a term project in Graduate Computer Architecture at Berkeley's Computer Science Division. Thanks for guidance and helpful feedback go to Prof. David Culler and Klaus-Eric Schauser.

Sather and this project would not have been possible without its designer and principal developer, Steve Omohundro.

References

[1] Mark D. Hill. Dinero III cache simulator. Software Distribution for *Computer Architecture: A Quantitative Approach.* Revised Version, August 1990.

[2] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[3] Stephen M. Omohundro. The Sather Language. Technical report, International Computer Science Institute, Berkeley, Ca., 1991.

[4] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, San Mateo, Ca., 1990.

[5] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[6] David Michael Ungar. The design and evaluation of a high performance smalltalk system. Technical Report UCB/CSD 86/287, Computer Science Division, University of California, Berkeley, Ca., March 1986.