# Pseudoknot: a Float-Intensive Benchmark for Functional Compilers (DRAFT)

Pieter H. Hartel [a] Marc Feeley [b] Martin Alt [c] Lennart Augustsson [d]
Peter Baumann [e] Marcel Beemster [f] Emmanuel Chailloux [g]
Christine H. Flood [h] Wolfgang Grieskamp [i] John H. G. van Groningen [j]
Kevin Hammond [k] Bogumił Hausman [l] Melody Y. Ivory [m] Peter Lee [n]
Xavier Leroy [o] Sandra Loosemore [p] Niklas Röjemo [q] Manuel Serrano [r]
Jean-Pierre Talpin [s] Jon Thackray [t] Pierre Weis [u] Peter Wentworth [v]

September 11, 1994

**Abstract**

Over 20 implementations of different functional languages are benchmarked using the same program, a floating-point intensive application taken from molecular biology. The principal aspects studied are compile time and execution time for the varying implementations. An important consideration is how the program can be modified and tuned to obtain maximal performance on each language implementation.

With few exceptions, the compilers take a significant amount of time to compile this program, though almost all compilers were faster than the current GNU C compiler. Compilers that generate C or Lisp are often slower than those that generate native code directly; the cost of compiling the intermediate form is normally a large fraction of the total compilation time.

There is no clear distinction between the runtime performance of eager and lazy implementations when appropriate annotations are used: lazy implementations have clearly come of age when it comes

[a] Dept. of Computer Systems, Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, e-mail: pieter@fwi.uva.nl

[b] Départ. d'informatique et r.o., univ. de Montréal, succursale centre-ville, Montréal H3C 3J7, Canada, e-mail: feeley@iro.umontreal.ca

[c] Informatik, Universität des Saarlandes, 66041 Saarbrcken 11, Germany, e-mail: alt@cs.uni-sb.de

[d] Dept. of Computer Systems, Chalmers Univ. of Technology, 412 96 Göteborg, Sweden, e-mail: augustss@cs.chalmers.se

[e] Dept. of Computer Science, Univ. of Zurich, Winterthurerstr. 190, 8057 Zurich, Switzerland, e-mail: baumann@ifi.unizh.ch

[f] Dept. of Computer Systems, Univ. of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, e-mail: beemster@fwi.uva.nl

[g] LIENS, URA 1327 du CNRS, École Normale Supérieure, 45 rue d'Ulm, 75230 PARIS Cédex 05, France, e-mail: Emmanuel.Chailloux@ens.fr

[h] Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge Massachusetts 02139, USA, e-mail: chf@lcs.mit.edu

[i] Berlin University of Technology, Franklinstr. 28-29, 10587 Berlin, Germany, e-mail: wg@cs.tu-berlin.de

[j] Faculty of Mathematics and Computer Science, Univ. of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, e-mail: johnvg@cs.kun.nl

[k] Dept. of Computing Science, Glasgow University, 17 Lilybank Gardens, Glasgow, G12 8QQ, UK, e-mail: kh@dcs.glasgow.ac.uk

[l] Computer Science Lab, Ellemtel Telecom Systems Labs, Box 1505, S-125 25 Älvsjö, Sweden, e-mail: bogdan@erix.ericsson.se

[m] Computer Research Group, Institute for Scientific Computer Research, Lawrence Livermore National Laboratory, P. O. Box 808 L-419, Livermore, CA 94550, e-mail: ivory1@llnl.gov

[n] Dept. of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue Pittsburgh, Pennsylvania 15213, USA, e-mail: petel@cs.cmu.edu

[o] INRIA Rocquencourt, projet Cristal, B.P. 105, 78153 Le Chesnay, France. e-mail: Xavier.Leroy@inria.fr

[p] Dept. of Computer Science, Yale Univ., New haven, Connecticut, e-mail: loosemore-sandra@cs.yale.edu

[q] Dept. of Computer Systems, Chalmers Univ. of Technology, 412 96 Göteborg, Sweden, e-mail: rojemo@cs.chalmers.se

[r] INRIA Rocquencourt, projet Icsla, B.P. 105, 78153 Le Chesnay, France. e-mail: Manuel.Serrano@inria.fr

[s] European Computer-Industry Research Centre, Arabella Strasse 17, D-81925 Munich, Germany. e-mail: jp@ecrc.de

[t] Harlequin Ltd, Barrington Hall, Barrington, Cambridge CB2 5RG, England, e-mail: jont@harlequin.co.uk

[u] INRIA Rocquencourt, projet Cristal, B.P. 105, 78153 Le Chesnay, France. e-mail: Pierre.Weis@inria.fr

[v] Dept. of Computer Science, Rhodes Univ., Grahamstown 6140, South Africa, e-mail: cspw@cs.ru.ac.za

to implementing largely strict applications, such as the pseudoknot program. The speed of C can be approached by some implementations, and is even exceeded by Sisal on the Cray, but in order to achieve this performance, special measures such as strictness annotations are required by non-strict implementations.

# 1 Introduction

At the Dagstuhl Workshop on Applications of Functional Programming in the Real World in May 1994 [17], several interesting applications of functional languages were presented. One of these applications, the pseudoknot problem [14] had been written in several languages, including C, Scheme [40], Multilisp [22] and Miranda[1] [50]. A number of workshop participants decided to test their compiler technology using this particular program. One point of comparison is the speed of compilation and the speed of the compiled program. Another important point is how the program can be modified and tuned to obtain maximal performance on each language implementation available. It is also interesting to compare the performance of typed and untyped languages. Finally, an interesting question is whether laziness is or is not beneficial for this application.

The initial benchmarking efforts revealed important differences between the various compilers. The first impression was that compilation speed should generally be improved. After the workshop we have continued to work on improving both the compilation and execution speed of the pseudoknot program. Some researchers not present at Dagstuhl joined the team and we present the results as a record of a small scale, but exciting collaboration with contributions from many parts of the world.

As is the case with any benchmarking work, our results should be taken with a grain of salt. We are using a realistic program that performs a useful computation, however it stresses particular language features that are probably not representative of the applications for which the language implementations were intended. Implementations invariably trade-off the performance of some programming feature for others in the quest for the right blend of usability, flexibility, and performance on "typical" applications. It is clear that a single benchmark is not a good way to measure the overall quality of an implementation. Moreover, the performance of an implementation usually (but not always) improves with new releases as the implementors repair bugs, add new features, and modify the compiler. We feel that our choice of benchmark can be justified by the fact that it is a real world application, that it had already been translated into C and several functional languages, and that we wanted to compare a wide range of languages and implementations. The main results agree with those found in earlier studies [8, 25].

Section 2 briefly characterises the functional languages that have been used. The pseudoknot application is introduced in Section 3. The compilers and interpreters for the functional languages are presented in Section 4. Section 5 describes the translations of the program from one language into the other. The results and conclusions are given in the last two sections.

# 2 Languages

Details of the languages which were benchmarked may be found in Table 1. C has been used as a reference language in order to allow comparison with imperative languages. The first column of the table gives the name of the language. The second column gives the source (i.e. a University or a Company) if a language has been developed in one particular place. Some languages were designed by a committee, which is also shown. The third column of the table gives a key reference to the language definition.

The last four columns describe some important properties of the languages. The typing discipline may be *strong* (and static), *dynamic*, or *weak*; a strong typing discipline may be monomorphic (*mono*), polymorphic (*poly*), or parametrically polymorphic (*param*). The evaluation strategy may be *eager*, *lazy* or *eager* with *non-strict* evaluation [34]. The language may be *first order* or *higher order*. Accessing components of data structures may be supported by either pattern-matching on function arguments, local definitions and/or as part of case expressions (*pattern*, *case*), by compiler generated access functions to destructure data (*access*), or not at all (*none*). The reader should consult the references provided for further details of individual languages.

[1] Miranda is a trademark of Research Software Ltd.

| language | source | ref. | typing | evaluation | order | match |
|---|---|---|---|---|---|---|
| Caml | INRIA | [53] | strong, poly | eager | higher | pattern |
| Clean | Nijmegen | [38] | strong, poly | lazy | higher | pattern |
| Common Lisp | Committee | [48] | dynamic | eager | higher | access |
| Erlang | Ericsson | [3] | dynamic | eager | first | pattern |
| Facile | ECRC | [49] | strong, poly | eager | higher | pattern |
| Gofer | Yale | [28] | strong, poly | lazy | higher | pattern |
| Haskell | Committee | [27] | strong, poly | lazy | higher | pattern |
| ID | MIT | [34] | strong, poly | eager [a] | higher | pattern |
| Miranda | Kent | [50] | strong, poly | lazy | higher | pattern |
| Opal | TU Berlin | [10] | strong, param | eager | higher | pattern |
| RUFL | Rhodes | [55] | strong, poly | lazy | higher | pattern |
| Scheme | Committee | [40] | dynamic | eager | higher | access |
| Sisal | LLNL | [32] | strong, mono | eager | first | none |
| Standard ML | Committee | [33] | strong, poly | eager | higher | pattern |
| Stoffel | Amsterdam | [5] | strong, poly | lazy | higher | case |
| Trafola | Saarbrücken | [1] | strong, poly | eager | higher | pattern |
| ANSI C | Committee | [29] | weak | eager | first | none |

[a] eager, non-strict

Table 1: Language characteristics. For each language its source is given, as well as a key reference to the language definition. The rest of the table presents important language characteristics.

| floating point operations | | square root and trigonometric functions | |
|---|---|---|---|
| $\times$ | 3,567,672 | | |
| $+$ | 2,798,571 | $\sqrt{\ }$ | 69,600 |
| $>, \leq, <$ | 129,656 | arctan | 40,184 |
| $-$ | 330,058 | cos | 40,184 |
| $/$ | 40,184 | sin | 40,184 |
| total | 6,866,141 | total | 190,152 |

Table 2: Breakdown of the "real" work involved in the pseudoknot problem as counted by the FAST system. The floating point operations occurring in the trigonometric functions and the square root are *not* counted separately.

# 3 Application

The pseudoknot program is derived from a "real-world" molecular biology application that computes the three-dimensional structure of part of a nucleic acid molecule. The program exhaustively searches a discrete space of shapes and returns the set of shapes that respect some structural constraint.

The program is heavily floating point oriented. For example the C version of the program spends 25% of its time in the C library trigonometric and square root routines.

Statistics from the lazy FAST [24] compiler show that with lazy evaluation the fastest version of the program does about 7 M floating point operations, which excludes those performed by the 190 K trigonometric and square root functions. A detailed breakdown of these statistics is shown in Table 2. The program makes about 1.5 M function calls and claims about 15 Mbytes of space (The maximum heap size required is about 30 Kbytes). Statistics from the eager MLWorks system show that with eager evaluation the program spends about 50% of its time performing floating point operations. All this suggests that the application can be characterised as somewhere in between "numeric" and "symbolic".

The pseudoknot program does not explicitly depend on laziness for its correct behaviour. The program is thus ideal for comparing lazy and non-lazy implementations of functional languages. It is relatively easy to translate the program from a lazy to a strict functional language. Inserting strictness annotations in a lazy version of the program is also relatively easy.

The program uses mostly algebraic data types as data structures. Some of these are rather large.

Having to use pattern-matching style destructuring on data structures with as many components as 12 or even 30 leads to distinctly unreadable code.

With the obvious exception of the C version, all other versions of the program are purely functional.

The original program is described in detail by Feeley, et. al [14]. The program used in the present benchmarking effort differs in two ways from the original.

Firstly, the original program only computed the number of solutions found during the search. However, it is the location of each atom in the solutions that are of interest to a biologist because these solutions typically need to be screened manually by visualizing them one after another. The program was thus modified to compute the location of each atom in the structures that are found. In order to minimize I/O overhead, a single value is printed: the distance from the origin to the farthest atom in any solution (this requires that the position of each atom be computed).

Secondly, the original program did not attempt to exploit laziness in any way. However, there is an opportunity for laziness in the computation of the absolute position of atoms. To compute the absolute position of an atom, it is necessary to transform a 3D vector from one coordinate system to another by multiplying a 3D vector by a $3 \times 4$ transformation matrix. The reason for this is that the position of an atom is expressed relatively to the nucleotide it is in. Thus the position of an atom is specified by two values: the 3D position of the atom in the nucleotide, and the absolute position and orientation of the nucleotide in space. However, the position of atoms in structures that are pruned by the search process are not needed (unless they are needed to test a constraint). There are thus two formulations of the pseudoknot program which differ in the way the position computation of atoms is expressed:

**Late formulation** The absolute position of an atom is computed each time it is needed; either for testing a constraint or for computing the distance to origin if it is in a solution. This formulation may duplicate work. Because the placement of a nucleotide is shared by all the structures generated in the subtree that stems from that point in the search, there can be as many recomputations of a position as there are nodes in the search tree (which can number in the thousands to millions).

**Early formulation** The absolute position of an atom is computed as soon as the nucleotide it is in is placed. If this computation is done lazily, this formulation avoids the duplication of work because the coordinate transformation for an atom is at most done once.

The original program used the late formulation. To explore the benefits of laziness, the program was modified so that it is easy to obtain the alternative early formulation (only a few lines need to be commented and uncommented to switch from one formulation to the other).

The C, Scheme, and Miranda versions of the program support the early formulation. The Miranda version implicitly uses lazy evaluation. The Scheme version uses the explicitly lazy *delay* and *force* constructs. The C version requires lazy computation to be explicitly programmed.

# 4 Compilers

An overview of the compilers which were studied may be found in Table 3. The first column gives the name of the language and/or compiler, the second shows the source of the compiler. A key reference that describes the compiler is given in the third column. The last column gives instructions for obtaining the compiler by FTP or email.

Most of the experiments were performed on Sun SPARC workstations. The basis of the experiments is formed by the performance of the C compiler. We have used either Sun's bundled C compiler cc or the GNU C compiler gcc. The gcc compiler generates slightly faster code for the pseudoknot program, but it takes about three times as long to compile. Table 4 gives the detailed results as measured on machine 7 (c.f. Table 13) using gcc version 2.5.8. These results indicate that the C version of the pseudoknot program is difficult to compile, even for a C compiler. This is due to the presence of four large routines (consisting of 574, 552, 585 and 541 lines of code, respectively) which together build the conformation data base of the nucleotides. Stripping the body of these four functions to be empty (which still leaves a program of 1073 lines of C code) reduces the compilation time to about 13 seconds, for cc as well as gcc. All the functional versions of the program have the same structure as the C version, so the functional compilers are faced with the same difficulty of compiling the code that builds the conformation data base.

An interesting difference between cc and gcc is that over 75% of the compilation time for cc is spent in the assembler, whereas gcc spends 99% of its time in the main compiler pass (`cc1`).

| language | version | source | ref. | FTP / Email |
|---|---|---|---|---|
| Bigloo | 1.7 | INRIA Rocquencourt | [44] | Ftp: ftp.inria.fr: /INRIA/Projects/icsla/Implementations/ |
| Caml Light | 0.61 | INRIA Rocquencourt | [12] | Ftp: ftp.inria.fr: /lang/caml-light/ |
| Caml Gallium | | INRIA Rocquencourt | [30] | Email: Xavier.Leroy@inria.fr |
| Camloo | 0.2 | INRIA Rocquencourt | [45] | Ftp: ftp.inria.fr: /lang/caml-light/ |
| CeML | 0.22 | LIENS | [9] | Email: Emmanuel.Chailloux@ens.fr |
| Clean | 1.0b | Nijmegen | [47] | Ftp: ftp.cs.kun.nl: /pub/Clean/ |
| CMU CL | 17e | Carnegie Mellon | [31] | Ftp: lisp-sun1.slisp.cs.cmu.edu: /pub/ |
| Erlang | 6.0.4 | Ellemtel AB | [26] | commercial Email: erlang@erix.ericsson.se |
| Facile | Antigua | ECRC | [49] | Email: facile@ecrc.de |
| FAST | 33 | Southampton/ Amsterdam | [24] | Email: pieter@fwi.uva.nl |
| Gofer | 2.30a | Yale | [28] | Ftp: nebula.cs.yale.edu: /pub/haskell/gofer/ |
| Haskell | 999.6 | Chalmers | [4] | Ftp: ftp.cs.chalmers.se: /pub/haskell/chalmers/ |
| Haskell | 0.22 | Glasgow | [36] | Ftp: ftp.dcs.glasgow.ac.uk: /pub/haskell/glasgow/ |
| Haskell | 2.1 | Yale | [21] | Ftp: nebula.cs.yale.edu: /pub/haskell/yale/ |
| ID | TL0 2.1 | MIT/Berkeley | [34] | Email: chf@lcs.mit.edu |
| MLWorks | n.a. | Harlequin Ltd. | [23] | commercial |
| Miranda | 2.018 | Research Software Ltd. | [51] | commercial Email: mira-request@ukc.ac.uk |
| Nearly Haskell | Pre rel. | Chalmers | [41] | Ftp: ftp.cs.chalmers.se: /pub/haskell/nhc/ |
| Opal | 2.1c | Berlin | [43] | Ftp: ftp.cs.tu-berlin.de: /pub/local/uebb/ocs |
| RUFL | 1.8.4 | Rhodes | [54] | Ftp: cs.ru.ac.za: /pub/rufl/ |
| Gambit Scheme | 2.2 | Montréal | [13] | Ftp: ftp.iro.umontreal.ca: /pub/parallele/gambit/ |
| Sisal | 12.9.2 | LLNL | [7] | Ftp: sisal.llnl.gov /pub/sisal |
| SML/NJ | 1.03z | AT&T Bell Labs. | [2] | Ftp: research.att.com: /dist/ml/ |
| Poly/ML | 2.05M | Abstract Hardware Ltd. | [15] | commercial |
| Stoffel | | Amsterdam | [6] | Email: beemster@fwi.uva.nl |
| Trafola | 1.2 | Saarbrücken | [1] | Email: alt@cs.uni-sb.de |

Table 3: Compiler details consisting of the name of the compiler and/or language, the University or Company that built the compiler, one key reference to the description of the implementation and the address from which the compiler can be obtained.

| C compiler | version | compilation complete | execution complete | compilation stripped |
|---|---|---|---|---|
| cc –O | SunOS 4.1.2 | 325 + 26 | 3.35 + 0.23 | 12.9 + 2.4 |
| gcc –O | 2.5.8 | 910 + 96 | 2.70 + 0.19 | 13.4 + 2.3 |
| ratio | | 0.35 + 0.26 | 1.23 + 1.15 | $\approx 1 + 1$ |

Table 4: The compilation and execution times (user+system seconds) of the C version of the pseudoknot program with the two most commonly used C compilers.

Table 5 shows some properties of the compilers and experiments that have an effect on the runtime performance of the implementations. In addition to the compile and runtime options (columns 2 and 3), the fourth column shows the type of garbage collector used, and the fifth column shows what floating point precision is used. The floating point precision is important to take into account when comparing results for two reasons. Firstly, using single precision floating point numbers makes it easier to generate fast code. Secondly, some architectures implement single precision floating point arithmetic significantly faster than double precision (DEC Alpha). However, on the SPARC that has been used for the majority of the experiments reported here there is only a small difference between the speed of single and double precision arithmetic.

# 5 Translation, annotations and optimisations

The pseudoknot program was translated by hand from the Scheme version to the various other languages. All versions were hand tuned to exhibit the best performance available with the compiler being used. The following set of guide lines has been used to make the comparison as fair as possible:

1. Algorithmic changes are forbidden but slight modifications to the code to better use a particular feature of the language or programming system are allowed. A modification that would be valid for all languages involved is not permitted.
2. Only small changes to the data structures are permitted (e.g. a tuple may be turned into an array or a list).
3. Annotations are permitted, for example strictness annotations, or annotations for inlining and specialisation of code.
4. All changes and annotations should be documented.
5. Anyone should be able to repeat the experiments. So all sources and measurement procedures should be made public (by `ftp` somewhere).
6. All programs must produce the same output (the number 33.7976 to 6 significant figures).

The optimisations and annotations made to obtain best performance with each of the compilers are discussed in the subsections to follow. Often we will make reference to particular parts of the program text. As the program is relatively large it would be difficult to reproduce it in full here, and in many versions. The reader is invited to consult the archive that contains most of the versions of the program at `ftp.fwi.uva.nl`, file `/pub/functional/pseudoknot.tar.Z`.

## 5.1 Bigloo

The Scheme version of the pseudoknot benchmark was changed only slightly to serve as input to the Bigloo optimising Scheme compiler. This compiler translates to C, as do many other compilers. In the C code it produces, the many vectors of floating point constants present in the conformation data base are not represented as ordinary C constants but as a large string. This enables the C compiler to compile the generated C much more quickly. During initialisation at runtime the string representation is transformed into the required numeric data.

| compiler | compiler options | execution options | collector | float |
|---|---|---|---|---|
| Bigloo | –unsafe –O4 | | mark-scan | double |
| Caml Light | | | gen. | double |
| Caml Gallium | | | gen. | double |
| Camloo | –unsafe –O4 | | mark-scan | double |
| CeML | –O | | 1-space | single |
| Chalmers | –c –Y–S –H50Mg –Y–A500k –cpp | | 2-space | single |
| Clean | | –nt –s 10k –h ... | 2-sp/ms | double |
| CMU | see text | see text | 2-space | single |
| Erlang BEAM | –fast | –h 600000 | 2-space | double |
| Facile | | | gen. | double |
| FAST | –fcg | –v 1 –h ... –s 400K 1 | 2-space | single |
| Gambit | –h14336 | –h14336 | 2-space | double |
| Glasgow | –O –fvia–C –O2–for–C | +RTS –H1M | gen. [42] | single |
| Gofer | –DTIMER | | 2-space | single |
| ID | strict, merge–partitions (tlc: opt) | | none | double |
| MLWorks | no details available[a] | | 2-space | double |
| Miranda | | /heap ...; /count | mark-scan | double |
| NHC(HBC) | –H30M | | 2-space | single |
| NHC(NHC) | –h2M | | 1-space | single |
| Opal | opt=full debug=no | | refcount | single |
| Poly/ML | –h 48000 | –h 48000 | gen. | single |
| RUFL | –w | –m300 | mark-scan | double |
| RUFLI | –iw | –m300 –r32000 | mark-scan | double |
| Sisal | –cpp –seq –O –c atan2 –cc=–O | | refcount | double |
| SML/NJ | | | gen. | double |
| Stoffel | –O2 (for C) | | 2-space | double |
| Trafola | –TC –INLINE 1 | –HE 8000000 | 1-space | sgl/dbl |
| Yale | see text | see text | 2-space | single |

[a] MLWorks is not yet available. Compilation was for maximum optimisation, no debugging or statistics collection was taking place at runtime.

Table 5: Compiler and execution options. The type of garbage collector is one of 2-space (non-generational 2-space copying collector); mark-scan; gen. (generational with two or more spaces); 1-space (mark-scan, one space compactor); or reference counting. Floating point arithmetic used is either single or double precision.

## 5.2 Caml

Three compilers for the Caml dialect of ML have been benchmarked: Caml Light, a simple bytecode compiler; Camloo, a Caml to C compiler derived from the Bigloo optimizing Scheme compiler; and Gallium, an experimental native-code compiler.

The source code is a straightforward translation to Caml of the SML version of the benchmark. The only changes made to increase performance was the use of an array instead of a list in the functions `list_of_atoms` and `most_distant_atoms`. For the Gallium compiler only, some tuples have been converted into records to guide the data representation heuristics; this transformation makes no difference for the other two compilers.

The Caml Gallium compiler uses unboxed double precision float in monomorphic parts of the source code. Since the pseudoknot benchmark does not use polymorphism, all floats are unboxed at all times. This is the main reason why the Gallium compiler generates faster code than most of the other compilers.

## 5.3 CeML

CeML is another dialect of Caml. The purpose of the development of CeML and its compiler is to prove that ML can be compiled into efficient C. CeML does not tag immediate values and uses a garbage collector with ambiguous roots which manages its own stack. This is important for the pseudoknot program because the floats are represented as a full word (32 bits).

The CeML source of the pseudoknot program differs only in one aspect from the Camloo source. To avoid rebuilding the tuple `(i,t,n)` the definition of `get_var` was changed from:

```
> fun get_var id ((i,t,n)::lst) = if id = i then (i,t,n)
>                                 else get_var id lst
```

to:

```
> fun get_var id (((i,_,_) as v)::lst) = if id = i then v
>                                        else get_var id lst
```

The total compilation time consist mainly of the gcc optimized compilation (–O2) time, particularly to build the data base. If the program is sliced into different modules and the module corresponding to the data base is not compiled with the –O2 option, the total compilation time is about ten minutes and the execution time does not change.

## 5.4 CLEAN

The syntax of the Clean 1.0 programming language is similar to the syntax of Haskell and Miranda. Porting the Miranda program to Clean 1.0 was easy. The guards were changed to Haskell syntax, the first character of each type name was changed to upper case, and the names of a few functions were changed. Some patterns on the right hand side of function definitions were moved to the left hand side or replaced by case expressions. This was necessary, because the current compiler cannot yet handle all patterns on the right hand side, only tuple and record patterns are currently allowed there. For example:

```
> anticodon_constraint v partial_inst
>  | i==33 = ...
>  where Var i t n = v
```

was replaced by:

```
> anticodon_constraint v=:(Var i t n) partial_inst
>  | i==33 = ...
>  where
```

To achieve good performance the following five changes have been made. The basic floating point operations and some constants were inlined. The components of the main algebraic data types `Pt` and `Tfo` and the integer component of `Var` were annotated as strict. The function `absolute_pos` was inlined in the function `atom_pos`. If a node on the right hand side of a definition occurred in the pattern, the code was changed to prevent rebuilding the node, for example:

```
> atom_pos atom (Var i t n) = absolute_pos (Var i t n) (atom n)
```

was changed to:

```
> atom_pos atom v=:(Var i t n) = absolute_pos v (atom n)
```

Finally, in the function `pseudoknot_domains` the values of `rA`, `rC`, `rG`, `rU` and `rCs` were assigned to local functions as: `ra=rA` etc. The local functions were used throughout the body of `pseudoknot_domains`. This is necessary because the current compiler does not yet support constant applicative forms (CAFs) [35].

## 5.5 Chalmers Haskell

The Haskell version of the pseudoknot program is a straightforward translation of the Miranda version. The latter often destructs and reconstructs a data item. In the Haskell version care has been taken to introduce "as" patterns thus:

```
> atom_pos atom v@(Var i t n) = absolute_pos v (atom n)
```

It proved necessary to split the program into a number of modules, as the program as a whole could not be compiled (see Section 5.14). Splitting the program into modules was straightforward since each of the four functions that initialise the conformation data base can be separated into its own large module of about 600 lines. The global data structure definitions are placed in their own module which is imported by each worker module, and the main program imports all of these modules.

The Chalmers Haskell compiler (HBC) provides a compiler option (–DSTR) to annotate all data structures as strict. The late formulation of the pseudoknot program runs about twice as fast with this option selected.

## 5.6 Erlang

Erlang is a concurrent functional programming language designed for prototyping and implementing reliable real-time systems [3]. The Erlang BEAM compiler [26] is an efficient and portable implementation of Erlang where Erlang programs are compiled into C. However, Erlang is a concurrent language and has explicit syntax for referring to time (time-outs). The implementation therefore has to provide a scheduling mechanism and a notion of time which introduces constant overhead to the Erlang BEAM execution.

Porting the benchmark from Miranda to Erlang required some minor changes. Since Erlang is not a higher order language some function calls were done by applying `p_apply/N`, for example `reference/3` is called as:

```
> p_apply(reference,Arg1,Arg2,Arg3) -> reference(Arg1,Arg2,Arg3).
```

when all arguments are known. Another problem is typing. Some limited typing can be done in Erlang using guard tests. For example, the following Erlang function types its arguments as floats:

```
> pt_sub({X1,Y1,Z1},{X2,Y2,Z2})
>       when float(X1),float(Y1),float(Z1),
>            float(X2),float(Y2),float(Z2) ->
>       {X1-X2,Y1-Y2,Z1-Z2}.
```

The type information obtained from guards and extended by compile-time type analysis is used by the Erlang BEAM compiler to optimize different arithmetic operations.

Since Erlang BEAM compiles Erlang programs into C, the Erlang BEAM compilation time presented in Table 15 includes the costs both of compiling Erlang to C and of calling gcc (version 2.5.8) to compile the generated C program. The total compilation time consists mainly of the gcc compilation time (4751 + 20 seconds). The gcc compiler was run with the -O2 option which increases compilation time but a better performance of the generated code was considered more important than the compilation time.

The best execution time was achieved by extending the available heap space to minimize the garbage collection time, and by inlining floating point operations after compile-time type analysis.

| version | unoptimised | optimised |
|---|---|---|
| curried | 8.48 + 0.51 | 8.38 + 0.51 |
| uncurried | 8.03 + 0.50 | 7.66 + 0.49 |

Table 6: Four versions of the pseudoknot program executed with the Facile Antigua system, showing execution times (user+system seconds).

Erlang BEAM provides double precision floating point arithmetic. All floating point numbers are stored as boxed objects in the heap. Integers are 32 bits objects, which may be stored directly on stack. The relatively poor performance of the Erlang BEAM compiler (reported in Table 16) is due to the fact that the pseudoknot benchmark is float-intensive and thus tests mainly the implementation of floating point arithmetic. As shown later in Section 5.15, having single precision floating point arithmetic and using an unboxed representation of floating point data may improve the execution time by a factor of 4. In the Erlang BEAM compiler giving up the double precision floating point arithmetic would similarly improve both access and garbage collection time by storing single-float objects directly on stack instead of heap.

To support the above claim we did the following simple experiment. We ran the two functions `tfo_apply/2` and `tfo_combine/2`, which are very floating-point intensive, using first integers and then (double precision) floats as input data. The experiment was carried out on machine 19from Table 13. The results showed that using integers instead of floats improved the Erlang BEAM execution time of those two functions by a factor of 6. To be sure that the improvement comes from better data representation and not from better integer arithmetic provided by the hardware, a simple C program was written to compare the speed of floating-point and integer arithmetic. The results showed that the floating-point arithmetic was as efficient as the integer arithmetic.

Since single precision floats can be represented in Erlang BEAM in the same way as integers we can expect that giving up the double precision floating point arithmetic would improve the pseudoknot execution time to a large extent.

## 5.7 Facile

Facile is a high-level, higher-order programming language for systems that require a combination of complex data manipulation and concurrent and distributed computing. It combines Standard ML (SML), with a model of higher-order concurrent processes based on CCS. Facile is well-suited for running on loosely connected, physically distributed systems with distributed memory. It is possible to execute Facile programs on both local area networks (LANs) and wide area networks (WANs).

The basic computational model of Facile consists of one or more nodes or virtual processors, on each of which there are zero or more processes (light weight threads). Processes execute by evaluating expressions, and they can communicate values between each other by synchronising over typed channels. All types of values, including user defined types, channels, functions and process scripts, are transmissible.

The pseudoknot benchmark does not use the distributed capabilities of the Facile system. Yet it is interesting to see that the cost of starting up a sequential program in the Facile system is small (26 milli seconds on machine 20(c.f. Table 13).

We did several experiments with the Pseudoknot benchmark program. The results are shown in Table 6. Firstly, the Standard ML version of the pseudoknot program was run unoptimised and unmodified (i.e. with curried function definitions). Secondly, the execution time was found to be unaffected by the following compiler optimisations: loop unrolling, scope localisation of free variables, hoisting, contraction, common subexpression elimination, range analysis, and inlining. Thirdly, uncurrying function definitions made the program slightly faster, and fourthly, provided more scope for the compiler optimisations. The Facile compiler is essentially based on the SML/NJ, version 0.93. In Section 5.18 we describe experiments with a more recent release of the SML/NJ compiler, which is about twice as fast for the pseudoknot benchmark.

| strictness | late | | early | |
|---|---|---|---|---|
| annotations | seconds | Mbytes | seconds | Mbytes |
| without | 31.8 + 0.7 | 89.5 | 30.1 + 4.9 | 94.1 |
| with | 11.0 + 0.5 | 15.5 | 12.1 + 1.2 | 35.3 |

Table 7: Four versions of the pseudoknot program executed with the (lazy) FAST system, showing execution times (user+system seconds) and heap allocation (Mbytes).

### 5.8 FAST

The FAST compiler input language is a subset of Miranda; the port from Miranda to this subset is trivial. Three changes have been made to achieve best performance. Firstly, all components of the main algebraic data types (`pt`, `tfo` and `var`) in the program were annotated as strict. Secondly, the basic definitions of floating point operations such as `fadd = (+)` and `fsin = sin` were inlined. Finally, one construct that the FAST compiler should be able to deal with, but which it cannot at present handle has been changed. Functions such as:

```
> atom_pos atom (Var i t n) = absolute_pos (Var i t n) (atom n)
```

were replaced by:

```
> atom_pos atom v = absolute_pos v (atom (var2nuc v))
> var2nuc (Var i t n) = n
```

To explore the possible advantage of laziness, a number of different versions of the pseudoknot program have been compiled, executed and analysed using the FAST system. Table 7 shows the total execution time (user+system seconds) and the total amount of heap space (Mbytes) consumed by four different versions of the program. Both the late and early formulations of the program were tested with and without strictness annotations. The strictness annotations declare the three primary data types (`pt`, `tfo` and `var`) of the program strict.

Strictness annotations have a profound effect on the performance of the pseudoknot program. The user time of the late formulation is reduced by a factor of 2.9 and the user time of the early formulation is reduced by a factor of 2.5. These improvements are due to the reduction in heap allocation. A less profound effect is caused by the laziness of the program. There are two differences between the early and late formulations of the pseudoknot program: the early formulation does fewer computations but it allocates more heap space. Regardless of whether strictness annotations are used or not, the number of floating point multiplications made by the early formulation is 19% less and the number of floating point additions is 14% less.

Without strictness annotations the early formulation allocates 5% more heap space than the late formulation. The combined effect of allocating a little more space, but doing less work makes the early formulation 9% faster than the late formulation. With strictness annotations the early formulation allocates 44% more space than the late formulation. This time the reduction in heap space more than cancels the savings in floating point operations. Now the late formulation is 10% faster than the early formulation.

### 5.9 Gambit

Gambit is an R4RS Scheme conformant optimizing compiler which supports the whole numeric tower (complex, real, rational, and integer). Measurements could not be done on the SPARC because currently the only fully complete code generator is for M680x0 processors. Since the original program had been developed with Gambit the source code was not modified in any way from the distributed source code ([14] gives the details of the development process and an analysis of the program when compiled with Gambit).

Both the late and early formulations were tested. Table 8 shows the total execution time for the program when compiled by Gambit 2.2 and gcc 2.0 on machine 8 (c.f. Table 13). On this benchmark, the code generated by Gambit executes at about 37% the speed of C. This slowdown is due in a large part to the fact that Gambit boxes all floating point numbers. To better evaluate this cost, the two

| Compiler | late | early |
|---|---|---|
| Gambit 2.2 | 17.30+0.22 | 25.90+0.35 |
| gcc 2.0 | 6.34+0.12 | 9.75+0.15 |

Table 8: Execution times (user+system seconds) for both formulations of the pseudoknot program when compiled with Gambit and gcc 2.0.

most numerically intensive functions (`tfo_combine` and `tfo_apply`) were hand coded in assembler to avoid the boxing of intermediate floating point values; the input and result of the functions were still boxed. The execution time of the late formulation went down by a substantial 34% to 11.4 seconds, that is 55.6% the speed of C. Since the two functions are relatively small and have a trivial data flow and control flow, a simple "local" type analysis would probably be sufficient for the compiler to generate much better code for this benchmark.

For both Gambit and gcc, the early formulation is roughly 50% slower than the late formulation. Thus, for the pseudoknot data set, the cost of lazy computation in the early formulation is larger than the saving in computation it provides. It is conceivable however that the early formulation can surpass the late formulation if the program is run with a data set which generates few solutions compared to the number of shapes searched. In such a case, a small proportion of the atom positions will have to be computed.

## 5.10 Glasgow Haskell

The Glasgow compiler is a highly-optimising Haskell compiler, with elaborate performance monitoring tools. The pseudoknot program was subjected to much optimisation on the basis of results from these tools.

The late formulation of the program (identical to the Chalmers Haskell program described in Section 5.5) ran in 10.2 seconds on machine 9 (c.f. Table 13). Based on the raw time profiling information from this program (Table 9-a), it is obvious that a few functions account for a significant percentage of the time used, and over 80% of the total space usage. Three of the top four functions by time (`tfo_combine`, `tfo_apply` and `tfo_align`) manipulate `TFO`s and `Pt`s, and the remainder are heavy users of the `Var` structure. Since these functions can be safely made strict, they are prime candidates for the use of unboxed types [37]; types whose values can be represented literally rather than by pointers to nodes in the heap. Unboxing is desirable when possible both because execution is faster and because a program's space requirements can generally be reduced.

Unboxing these data structures semi-automatically, and strictifying the lazy pattern match in the definition of `var_most_distant_atom` gives a significant performance improvement, of roughly a factor of 3. This is similar to the improvements which are possible by simply annotating the relevant data structures to make them strict. However, further unboxing optimisations are possible if the three uses of the function composition `maximum . map` are replaced by a new function `maximum_map` as shown below. This function maps a function whose result is a floating-point number over a list of arguments, and selects the maximum result. It is not possible to map a function directly over a list of unboxed values using the normal Prelude `map` function, because unboxed values cannot be passed to polymorphic functions.

```
> maximum_map :: (a->Float#) -> [a]->Float#
> maximum_map f (h:t) =
>    max f t (f h)
>    where max f (x:xs) m = max f xs (let fx = f x in
>                                     if fx `gtFloat#` m then fx else m)
>          max f [] m = m
>          max :: (a->Float#) -> [a] -> Float# -> Float#
```

This optimisation is suggested indirectly by the time profile (Table 9-b) which shows that the top function by time is `tfo_apply`. This is called through `absolute_pos` within `most_distant_atom`. It seems likely that the three nested calls to produce the maximum value of a function over a list of values

| **Cost Centre** | %time | %alloc |
|---|---|---|
| tfo_combine | 18.0 | 4.7 |
| tfo_apply | 15.9 | 0.0 |
| p_o3' | 8.3 | 25.5 |
| tfo_align | 6.2 | 1.9 |
| dgf_base | 5.9 | 21.6 |
| get_var | 5.9 | 0.0 |
| absolute_pos | 4.7 | 24.1 |
| ... | ... | ... |

(a) Original profile (by time)

| **Cost Centre** | %time | %alloc |
|---|---|---|
| tfo_apply | 11.1 | 0.0 |
| tfo_combine | 10.5 | 23.2 |
| search | 9.9 | 2.3 |
| pseudoknot_constraint | 8.2 | 8.7 |
| get_var | 7.0 | 0.0 |
| var_most_distant | 6.4 | 8.7 |
| tfo_align | 5.8 | 4.9 |
| ... | ... | ... |

(b) Strict types (by time)

| **Cost Centre** | %time | %alloc |
|---|---|---|
| get_var | 11.1 | 0.0 |
| tfo_combine | 10.5 | 26.5 |
| p_o3' | 8.5 | 13.0 |
| pseudoknot_constraint | 7.8 | 9.9 |
| search | 7.8 | 2.6 |
| tfo_align | 5.2 | 5.6 |
| pt_phi | 5.2 | 0.0 |
| tfo_apply | 5.2 | 0.0 |
| ... | ... | ... |

(c) Maximum map (by time)

| **Cost Centre** | %time | %alloc |
|---|---|---|
| tfo_combine | 9.7 | 26.5 |
| p_o3' | 7.7 | 13.0 |
| pseudoknot_constraint | 4.6 | 9.9 |
| mk_var | 2.0 | 6.6 |
| tfo_align | 10.2 | 5.6 |
| tfo_inv_ortho | 2.6 | 5.6 |
| ... | ... | ... |

(d) Maximum map (by allocation)

Table 9: Time and allocation profile by function, as a percentage of total time/heap allocations.

| Version | late | | | early | | |
|---|---|---|---|---|---|---|
| | seconds | Mbytes | (Residency) | seconds | Mbytes | (Residency) |
| Original | 10.0 + 0.2 | 36.8 | (55K) | 11.0 + 0.3 | 57.6 | (419K) |
| Strict Types | 3.5 + 0.3 | 10.1 | (53K) | 3.4 + 0.2 | 30.6 | (215K) |
| Maximum Map | 1.8 + 0.1 | 7.6 | (46K) | 2.9 + 0.1 | 29.3 | (215K) |

Table 10: Time and Heap Usage of three pseudoknot variants compiled for machine 9 by the Glasgow Haskell compiler.

can be merged to hold the current maximum in a register (an extreme form of deforestation [52]). This optimisation reduces the total execution time to 1.8 seconds user time.

The final time profile (Table 9-c) shows `get_var` and `p_o3'` jointly using 20% of the Haskell execution time with `tfo_combine`, `tfo_align` and `tfo_apply` accounting for a further 20%. (The minor differences in percentage time for `tfo_combine` in Tables 9-c and 9-d are probably explained by sampling error over such a short run.) While the first two functions could be optimised to use a non-list data structure, it is not easy to optimise the latter functions any further. Since the total execution time is now close to that for C, with a large fraction of the total remaining time being spent in the Unix mathematical library, and since the allocation profile (Table 9-d) also suggests that there are no space gains which can be obtained easily, it was decided not to attempt further optimisations. The Glasgow Haskell overall time and space results are summarised in Table 10. The heap usage is the total number of bytes allocated in each case, with the maximum live data residency after a major garbage collection shown in parentheses.

As with Erlang and other C-based compilers, much of the compilation time is spent in the C compiler. A native code generator is available, and is faster, but this does not allow all the optimisations which were attempted.

It is planned to incorporate an automatic generalised version of the deforesting optimisation, the foldr/build transformation [18], into the Glasgow Haskell compiler in due course.

## 5.11 Gofer

The Gofer version of the program was derived very quickly from the Miranda version supplied. The only significant changes were needed to accommodate the different notations for infix operators and comments. The Gofer primitive `atan2` function was used in place of the `math_atan2` function in the Miranda code. Using `math_atan2` instead of the primitive `atan2` adds only 4% to the total execution time.

Gofer does not use any significant optimizations, nor does it provide a mechanism for annotating data type or function definitions to indicate, for example, strictness properties. No attempts were made to try to optimize the program to run faster or more efficiently in Gofer.

## 5.12 ID

ID is an eager, non-strict, mostly functional, implicitly parallel language. Only the purely functional subset of ID was used in this benchmark. The compiler used in this study was developed jointly between MIT and Berkeley. This first part of the compiler generates dataflow graphs from ID source code. The Berkeley back end compiles these dataflow graphs into Tl0 which is an abstract multi-threaded machine [19]. Then Tl0 is compiled into C. All of the other ID compilers are targeted to dataflow machines. The Berkeley back end creates separate files for each ID function, which contributes to the long compile time for ID.

The port of the pseudoknot benchmark from Haskell to ID was trivial. To improve the performance four changes were made. A number of the lists were changed to arrays; the recursive functions `get_var` and `search` were changed to be iterative using ID while loops; the local function `generate` within `p_03'` was replaced by an ID array comprehension; deforestation was applied (by hand) to the function composition `maximum . map`, as with the Glasgow compiler (Section 5.10). These changes cut the execution time roughly in half. They also reduced the space required for execution. There is associated overhead with compiling for parallel machines that we pay even on sequential machines. We compiled with –O2 and linked all of the executables using –static to improve performance.

## 5.13 MLWorks

The initial SML version of the pseudoknot program is not quite Standard ML. It contains use of a pervasive list length function, and an overloaded print function, neither of which are part of the pervasive environment provided by the language. These problems were corrected.

A list length function was provided in the obvious manner (i.e. tail recursive accumulating). The uses of the overloaded print function (two of) were replaced by calls to the relevant functions (output for the string and an MLWorks real printing function for the real).

An initial profile of the run of this program suggests that slightly over 50% of the run time is taken up in three functions, `tfo_combine`, `tfo_align` and `tfo_apply`. These do little more than floating point arithmetic and trigonometric functions, which means that half the time, little more than the floating point capability of the implementations is being tested. Some of that is functionality provided by a runtime library implemented through `libc`.

ML is a strict language. Thus if a function is defined but only used in one half of a conditional, a closure must still be built for that function before evaluating the conditional. Building a closure, for MLWorks, involves allocating heap and copying in values. Moving the function definition within the conditional avoids unnecessary closure building (which would be avoided anyway by a lazy implementation). Small savings were achieved by modifying the function `pseudoknot_constraint` to build `dist` only when required. The function `try_assignments` has been modified to be tail recursive.

## 5.14 NHC

The name "NHC" comes from the description "Nearly a Haskell Compiler". Speed is not the main goal of NHC, instead NHC is written so that it can recompile itself in less than 3 Mbytes of memory. Floating point performance is completely ignored, and all arithmetic operations are treated as user defined functions.

Two versions of the NHC compiler have been used. The first is NHC compiled by HBC, denoted as NHC(HBC). This is a relatively fast, but space hungry version of the compiler. The second version is NHC compiled by itself, denoted as NHC(NHC). This version is slower, but more economic in space.

The version of the pseudoknot program as it has been used with the HBC compiler has been changed for NHC in two ways.

Firstly, all six separate modules were combined into one module. This was done to show that NHC indeed compiles large programs using less memory than the other Haskell compilers. NHC(NHC) needs an 8 Mbytes heap to compile the single module version of the pseudoknot program, HBC could not compile the single module version of the pseudoknot program with 80 Mbytes of heap space, not even when using a single space garbage collector. NHC(HBC) needs 30 Mbytes of heap to compile the single module version of the pseudoknot program.

Secondly, all floating point types have been changed from `Float` to `Double`. This does not increase the precision as `Double` is implemented as single precision floating point in NHC, but it is necessary for the type checker.

The early formulation of the pseudoknot program is slightly faster than the late formulation, 170 seconds compared to 175 seconds (both on machine 3 c.f. Table 13). A possible explanation is that the early formulation does fewer arithmetic computations than the late formulation, and that NHC is bad at arithmetic.

## 5.15 OPAL

OPAL is based on language constructs similar to those found in most other functional languages. The port of the pseudoknot program was a matter of applying regular expressions to the SML and Haskell versions of the program. Some specialities of OPAL had to be tackled: (1) OPAL provides only strings and booleans as builtin data types, such that constant data has to be denoted by applying conversion functions to strings, and (2) OPAL requires an explicit type signature for each global function. Furthermore, a static limit on the number of components of a constructor made it necessary to split the common atoms of the data type `nuc` into two subrecords.

Only the late formulation of the pseudoknot program has been ported. No optimizing annotations have been added to the source. OPAL is strict by default, and the compiler performs inlining of function definitions across module boundaries automatically. Hence, in the target code, all floating point operations are inlined on the instruction level.

To measure the costs incurred in allocating floating point numbers in the heap, a new single precision floating point data type has been added to the system. These single precision numbers are represented as "unboxed" values, which thus do not incur garbage collection costs. The first two rows of Table 11 show the difference, as measured on machine 12 (c.f. Table 13). Arithmetic is still performed with double precision, but the results of the operations are stored with single precision. The timings suggest, that dynamically allocating floating point numbers slows down the pseudoknot program by about a factor of 2. The amount of heap space allocated does not differ much in both versions; this is a result of the reference counting based garbage collection scheme of the OPAL system, which minimizes the total amount of allocated memory at the expense of some execution speed.

The single precision floating point data type has been further refined by encapsulating values which do not require dynamic type information. These special values are not visible on the user level. However, after automatic unfolding and simplifications performed by the compiler, intermediate results of floating point operations are now passed literally to subsequent operations in the resulting code. This is similar to the way unboxed values are handled by the Glasgow Haskell compiler [37]. Unboxing intermediate results improves the execution time again by a factor of 2. The results are shown in the last row of Table 11.

The current distributed version 2.1b of the OPAL compilation system failed to properly compile the benchmark. The reason was that this compiler version packs all initialization code of a module into a single C function (C is the target language), which is – with several thousand lines of code for the nucleotide conformation data base – too large for the C compiler. The compilation scheme has been slightly modified to fix this design bug. The new scheme will be available in the next release 2.1c. As with other C-based compilers, more than 50% of the compilation time for the pseudoknot program is used by the GNU C compiler. The compilation is particularly slowed down by the need to check and compile several thousand applications of overloaded conversion functions from strings to numbers. This overhead is a result of the fact that numbers cannot be denoted literally in the OPAL language.

| version | time(s) | space(M) |
|---|---|---|
| double precision (boxed) | 19.9 + 0.26 | 1.10 |
| single precision (unboxed) | 9.7 + 0.23 | 0.85 |
| no intermediate type tags | 5.1 + 0.14 | 0.85 |

Table 11: Three versions of the pseudoknot program executed with the OPAL system on machine 12.

### 5.16 RUFL

RUFL is (almost) a subset of Haskell. It excludes type classes, and type signatures are mandatory for all top-level functions. It has an abridged module mechanism.

The source was ported from the modularized Chalmers Haskell version. The bulk of the effort went into providing the type signatures and module interface files. The main algebraic data types were renamed to avoid ambiguity between similarly named types and constructors. One function (`make_relative_nuc`) was too complex for the compiler and had to be split into two parts, one of the library primitives was named differently, and a top-level `main` function which does the output replaced the Haskell construct. No annotations were made. (The RUFL system does not support annotating components of data structures as strict.)

The compiler generates SPARC assembly code or SECD-like intermediate code which is interpreted. Timings for the interpreted version are denoted here as RUFLI. In both cases run-time memory organization is rather simplistically based around fixed-size cells, and algebraic data types are instantiated as chains of these cells. The heavy use of high-arity constructors in this example generates long chains and incurs considerable run-time penalties.

### 5.17 Sisal

Collaborators at Lawrence Livermore National Laboratory and Colorado State University developed the Sisal parallel functional language with the objective of obtaining high performance on commercial scalar and vector multiprocessors. Currently, mature Sisal compilers exist for a variety of shared memory scalar and vector multiprocessors. The two most important optimizations of the Sisal compiler are memory preallocation [39] and copy elimination [20]. These optimizations enable most Sisal programs to execute in place without copying; thus Sisal programs typically run as fast as equivalent imperative programs.

We ported the C version of the pseudoknot program to Sisal and made the following minor changes to increase performance. First, we converted the nucleotide database to an array-based structure instead of the record-based structure as in the C code. Sisal supports records, but the compiler is heavily optimized for arrays and array operations. Then, we rewrote the distance and transformation functions as scalar functions to eliminate array allocations and deallocations in the inner loops. Although array allocations and deallocations in Sisal are efficient (cost is constant for arrays of known size), it is difficult to recuperate the cost for small arrays.

Finally, we developed a recursive program that single threads the `partial_inst` stack and list of solutions. To accomplish this, we changed the code of the function `try` as follows:

| C | Sisal revised |
|---|---|
| | `> let` |
| add new element to stack | `>  stack1 := array_addh(stack)` |
| increment stack counter | `>  stack2 := pseudoknot_domains(stack1)` |
| pseudoknot_domains | `> in` |
| decrement stack counter | `>  array_remh(stack2)` |
| | `> end let` |

In principle, this code is identical to the C code. The Sisal compiler realizes that there is only a single consumer of each stack. It tags the data structure as mutable and generates code to perform all updates in place. Consequently, the Sisal code maintains a single stack structure similar to the C code, eliminating excessive memory usage and copy operations. The Sisal code runs in approximately 85KB of memory and achieves execution speeds comparable to the C code.

## 5.18 SML/NJ and Poly/ML

Both Standard ML of New Jersey and Poly/ML are implementations of Standard ML as described in the Definition of Standard ML [33]. The SML version of the pseudoknot application was initially derived from the Scheme version of the program (by Marc Feeley) and then subsequently modified to make better use of SML's typing facilities. These modifications had no effect on the execution time of the program, but improved its readability and robustness, and this was helpful for later experimentation with the code.

For SML/NJ, a signature was used to constrain the program (which was written as one large SML module) to export only the top-level `run` function. This had the effect of improving the compilation time as well as providing the compiler with more opportunities for automatic inlining and other optimizations. Other than that, no other changes were made to the program (though various things were tried, as explained below).

Initially, the program was compiled with versions 0.75 and 0.93 of the SML/NJ system. It was expected that the latest version (1.03z) would show a great improvement, since it employs a representation analysis [30], implemented by Zhong Shao (described in his Ph.D. thesis [46]), which would "unbox" many of the floating-point numbers stored in the rather large tuple data structures. Surprisingly, the first attempt with this version of the compiler resulted in a slowdown. This was due to the fact that the implementation of the representation analysis was not finished. In particular, unboxing of records of floating-point numbers was not yet supported, nor were floats being passed to functions in floating-point registers. When these features were completed (by Shao), the speed improved by more than a factor of two. Analysis of the compiler output for the pseudoknot program also showed that excessive register-spilling was occurring. A small improvement to the spilling heuristic (again by Shao) led to a further speedup of the code. The SML/NJ 1.03z compiler does not use FPU load and store instructions for loading and storing floating-point numbers. It is strongly suspected that changing this would lead to yet more large improvements to the running time.

Like the other functional versions of the pseudoknot program (including the Haskell and CLEAN versions), the SML version is "pure" in the sense that all of the data structures are immutable. (The SML program is also evaluated strictly; note, however, that the Haskell and CLEAN versions also use mostly strict evaluation.) Significant improvements might be possible if mutable arrays were used, in a manner similar to the C version of the program. However, one suspects that such a program would be less reliable and understandable than the current version.

A large number of variations on the SML program were tried, in an attempt to find better performance. The execution time was surprisingly stable under these changes, and in fact no change made any significant difference (good or bad) to the execution speed. Almost all of the functions in the program are presently curried; uncurrying made a barely measurable improvement. All of the various "deforestation" transformations used by the other implementations were also tried, and again none made any measurable improvement (though several led to minor slowdowns). It was recognized early on that the representation of the "tfo" matrix as a large tuple of values leads to a considerable amount of register spilling in the code generated by SML/NJ. However, changing the representation to an array of floating-point numbers made no improvement. The current version of the program also uses the late formulation for position computation of atoms; changing this to the early formulation greatly increased both the time and space required. In the end, the original transcription of the Scheme program, with the signature constraint, was used for all of the measurements.

The SML/NJ implementation of the pseudoknot program performs better on the DECstation 5000 than on the SPARC. On the DECstation 5000 it runs at 55% of the speed of C, on the SPARC it runs at only 36% of the speed of C. We suspect that this is mainly due to memory effects. Previous studies [11] have shown that the intensive heap allocation which is characteristic of the SML/NJ interacts badly with memory subsystems that use a write-no-allocate cache policy, as is the case of the SPARC; in contrast, the use of a write-allocate policy coupled with what amounts to sub-block placement on the DECstation (the cache block size is four bytes) supports such intensive heap allocation extremely well.

## 5.19 Stoffel

The Stoffel language is an intermediate language designed for studying code generation of high level languages to fine-grain parallel processors (see, for example, Fisher [16]). Because of this, the implementation is robust but lacks the sophistication of some of the other compilers discussed here. The source

| floats | compilation (user time) | execution | garbage collection |
|---|---|---|---|
| Single | 6.16 | 162.6 + 2.4 | 4% |
| Double | 6.35 | 186.5 + 5.7 | 11% |

Table 12: Trafola results comparing single and double precision arithmetic.

code that has been used is the same as that used for the FAST compiler (see Section 5.8), but without the strictness annotations. The intermediate format used by the FAST compiler has been translated automatically into Stoffel by the FAST compiler front-end.

The Stoffel compiler is based on a spineless abstract machine and implements many of the optimization commonly found, such as strictness and boxing analysis. The implementation uses double precision floating point numbers, which can also be passed around in unboxed form.

Most discouraging were the compilation times. The actual Stoffel compiler takes about 216 seconds to run, the gcc-compiler backend with –O2 takes more than two hours on machine 17 (c.f. Table 13). Most of this time is spent in compiling the function that initialises the floating point number representations.

Both the late and early formulations of the program were run *out of the box*, there was no tweaking. The late formulation runs 30% faster than the early formulation, the difference being relatively small because no strictness annotations have been used.

## 5.20 Trafola

The Trafola language and system has been designed as a (pure functional) transformation language in a compiler construction project. In addition to the usual functional constructs Trafola offers non-deterministic pattern matching. This allows for an expressive coding of algorithms, in particular tree transformations. The first version of the system and the language were untyped. User defined data types and Milner/Cardelli type inference were added later. At the moment there is still only one Trafola compiler and runtime system. This compiler is able to generate better code from typed programs than from untyped programs. The runtime system has to support both, at the cost of some efficiency.

The pseudoknot program was transformed from Miranda to typed Trafola basically by changing syntax. No special Trafola features were introduced. Basic definitions such as `fadd = (+)` were inlined and taken from the C library. The fact that Trafola was first designed as an untyped language, together with the provision of powerful pattern matching facilities, results in an unusual implementation of algebraic data types. Each $n$-ary constructor is represented as an $n+1$-tuple. The first component of the tuple is an element of an enumeration type e.g. `Foo a b c` is implemented as `(Foo,a,b,c)`.

The Trafola version of the pseudoknot program was compiled and executed using both single and double precision arithmetic. The results are shown in Table 12. Using double precision arithmetic increases compilation time because the internal representation of double precision floats is less compact, and because constant-folding also takes longer for doubles. At runtime, double precision floats are stored in the heap, whereas single precision values are stored in the stack. The double precision version therefore allocates more space and increases both execution and garbage collection times.

## 5.21 Yale Haskell

The Yale Haskell compiler uses Common Lisp as an intermediate language and relies on the Common Lisp compiler for translation to machine code. The version of Yale Haskell tested here uses CMU Common Lisp as its back end, although it also runs under most other Common Lisp implementations.

To prepare the pseudoknot program for running in Yale Haskell, all of the data structures were annotated as strict. (Internally, Yale Haskell represents the data structures as Common Lisp simple vectors containing tagged objects.)

The first argument of `get_var` and the arguments of `make_relative_nuc` have been forced to be strict because the case where these arguments are not used is an error. The functions `is_A`, `is_C`, and `is_G` have been inlined because they are small; this probably has no effect on timings. The functions `atom_pos` and `search` have been inlined to avoid a higher-order function call. These were the only changes made to the Chalmers Haskell version of the program; the actual code was unchanged.

The code was compiled with all optimizations enabled at the Haskell level. At the Lisp level, the same optimizations were used as for obtaining the CMU Common Lisp results. The program was run with a heap size of about 14 megabytes. The late formulation of the pseudoknot program runs more than twice as fast as the early formulation.

### 5.22 CMU Common Lisp

CMU Common Lisp is a public-domain implementation featuring a high-quality optimizing compiler. This compiler does a higher level of static type checking and type inference than most other Common Lisp compilers.

In the Common Lisp version of the program, the `pt` and `tfo` data types were implemented as vectors specialized to hold untagged `single-float` objects, rather than as general vectors of tagged objects. Other data types were implemented using the `defstruct` facility. Type declarations were added to all of the floating-point arithmetic operations and local variables that hold floating-point numbers. Otherwise, the code was a straight translation of the program from Scheme syntax. The resulting code is written in completely standard Common Lisp and uses no annotations or extensions specific to the CMU implementation.

The program was compiled with optimizations `(speed 3) (safety 0) (debug 0) (compilation-speed 0)`. CMU Lisp's block compilation feature was not used. The program was run with a heap size of about 14 megabytes.

The difference between the Yale Haskell and Common Lisp results as shown in Table 16 is due partly to use of tagged versus untagged arrays, and partly to the overhead of lazy lists in Haskell. These were the only significant differences between the hand-written Common Lisp code and the Lisp code produced by the Yale Haskell compiler. The Haskell code generator could be extended to use untagged arrays for homogeneous floating-point tuple types as well, but this has not yet been implemented.

## 6 Results

The pseudoknot program has always been compiled with option settings that should give fast execution, we have consistently tried to optimise for execution speed. The compile time and run time options used are shown in Table 5. To achieve best performance, no debugging, run time checks or profiling code has been generated. Where a "–O" option or higher optimisation setting could be used to generate faster code, we have done so.

We have tried to use one and the same machine where possible, but not all programs could be compiled and/or executed on the same machine, either because of commercial considerations, or because of the lack of a SPARC code generator. An overview of the machines used may be found in Table 13.

To factor out architectural differences, the C version of the pseudoknot program has been timed on all the machines involved. The execution time of the C version serves as the basic unit of time. The unit "pseudoknot" gives the relative speed with respect to C execution. It is computed as:

$$\text{relative speed} = \frac{100 \times \text{C execution time}}{\text{absolute time}}$$

To "run at 100 knots" thus means to run at exactly the same speed as C, and a speed of 1 knot is 1% of the speed of C.

The functional language compilers generate code that requires more memory than the code generated by the C compilers. Therefore, the functional implementations require larger caches to accommodate the same fraction of the code and data as the C implementation. This has a measurable effect on the pseudoknots. For example, we found the pseudoknots for the Sisal version of the benchmark program to increase with the cache size. Table 14 shows the speed of the Sisal version of the pseudoknot program relative to that of C on three machines with different cache sizes. We could have used a machine with a cache larger than 64K for all experiments to make the results look better. We have chosen not to do so because it is important to remember that using large amounts of memory is an important cost factor.

The "pseudoknot" as defined above is a relative measure. In the long run an absolute measure is also useful both as a measure of overall system performance, and to demonstrate absolute improvements in functional compiler technology independently of improvements in the C compilers (c.f. Drhystone, Whetstone). This requires choosing a particular architecture and C compiler as a reference to represent

| no. | machine | mem. | cache | op. system | processor | C compiler |
|---|---|---|---|---|---|---|
| 1 | SUN 4/670 | 64 M | 64 K | SunOS 4.1.3. | standard | gcc 2.5.7 |
| 2 | DECstat., 5000/200 | 32 M | 64 K | Ultrix 4.1. | standard | cc –O2 |
| 3 | SUN SPARC 10/30 | 32 M | 1 M | SunOS 4.1.3. | TMS390Z55 | gcc 2.5.8 |
| 4 | SUN 4/670 | 64 M | 64 K | SunOS 4.1.2. | Cypress CY605 | gcc 2.4.5 |
| 5 | SUN 4/690MP | 64 M | 64 K | SunOS 4.1.3 | ROSS 40MHz Super | cc |
| 6 | SUN 4/50 | 32 M | 64 K | SunOS 4.1.3. | standard | gcc 2.5.8 |
| 7 | SUN 4/690 | 64 M | 64 K | SunOS 4.1.2. | standard | gcc 2.5.8 |
| 8 | HP9000/385 | 32 M | 4K/4K | HP-UX B.09.00 | 68040 | gcc 2.0 |
| 9 | SUN SPARC 10/41 | 96 M | 20K/32K+1M | SunOS 4.1.3. | TMS390Z50 | gcc 2.5.7 |
| 10 | SUN SPARC 10/41 | 96 M | 1 M | SunOS 4.1.3. | standard | gcc 2.5.8 |
| 11 | SUN 4/330 | 96 M | | SunOS 4.1.1. | standard | gcc 2.5.4 |
| 12 | SUN 4/75 | 64 M | 64 K | SunOS 4.1.3. | standard | gcc 2.5.8 |
| 13 | SUN 4/690 | 64 M | 1 M | SunOS 4.1.3. | standard | gcc 2.5.8 |
| 14 | SUN 4/670MP | 64 M | 1 M | SunOS 4.1.3. | SUNW, system 600 | gcc 2.5.8 |
| 15 | CRAY C90 | 1024 M | none | UNICOS 7.C | custom vector | scc 4.0 |
| 16 | DECstat. 5000/200 | 128 M | 64 K | Mach 2.6 | standard | gcc 2.4 |
| 17 | SUN SPARC 10/41 | 128 M | 20K/32K+1M | Solaris 2.3 | standard | gcc 2.5.8 |
| 18 | SUN 4/670 | 64 M | 64 K | SunOS 4.1.3. | Cypress CY605 | gcc 2.4.5 |
| 19 | SUN SPARC 5 | 32 M | 24 K | SunOS 4.1.3. | standard (85 MHZ) | gcc 2.5.8 |
| 20 | SUN Sparc 10/41 | 64 M | 1M | SunOS 4.1.3. | standard | gcc 2.5.7 |

Table 13: Details of the machines and C compilers used to compile and/or execute the pseudoknot program. The make and type of the machine is followed by the size of the memory (in MB) the size of the cache (as a total or as instruction/data + secondary cache size), the operating system name and version, and the type of processor used. The last column gives the C compiler that has been used on the machine.

| machine | cache size | pseudoknots |
|---|---|---|
| 7 | 64KB | 73% |
| 17 | 20K/32K+1M | 92% |
| 15 | 36K+1M | 100% |

Table 14: The relative performance of the Sisal version as a function of the cache size.

the absolute 100% mark. Most execution times have been obtained on machine 7 (c.f. Table 13), using GCC version 2.5.8. This combination is therefore rather a convenient choice to deliver the absolute 100% pseudoknot. The C execution times for this machine are 2.71 seconds user time and 0.20 seconds system time.

To measure the times required by the faster programs with a reasonable degree of accuracy, the programs have been timed in a Unix C-shell loop as follows `time repeat 10 a.out`, or even `time repeat 100 a.out`. The resulting system and user times divided by 10 (100) are reported in the Tables 15 and 16, sorted by relative pseudoknot ratings on the user times.

## 6.1 Compilation

Table 15 shows the results of compiling the programs. The first column of the table shows the name of the compiler (c.f. Table 3). The second column "route" indicates whether the compiler produces native code ("N"), code for a special interpreter ("I"), or compiles to native code through a portable C back-end ("C"), Lisp ("L"), or Scheme ("S"), or through a combination of these back-ends. The third column gives a reference to the particular machine used for compilation (c.f. Table 13). The next two columns give the time and space required to compile the pseudoknot program. Unless noted otherwise, the space is the largest amount of space required by the compiler, as obtained from `ps -v` under the heading SIZE. The column marked "C-time" gives the time required to execute the C version of the pseudoknot program on the same machine as the one used for compilation. The last two columns "pseudoknots" show the relative and absolute performance with respect to C.

It is possible to distinguish broad groups within the compilers. The fastest compilers are, unsurprisingly, those that compile to an intermediate code for an interpreter, and which therefore perform few, if any, optimisations. NHC is an outlier, perhaps because unlike the other interpreters, it is a bootstrapping compiler. With the exception of Bigloo, which is faster than many native compilers, compilers that generate C are the slowest compilers. Not only does it take extra time to produce and parse the C, but C compilers have particular difficulty compiling code that contains large numbers of constants. As noted in Section 4, C compilers also have difficulty compiling the hand written C version of the pseudoknot program due to this phenomenon. The faster compilers also generally allocate less space. This may be because the slower compilers generally apply more sophisticated (and therefore space-intensive) optimisations.

## 6.2 Execution

All programs have been executed a number of times, with different heap sizes to optimise for speed. The results reported in Table 16 correspond to the fastest execution. This includes garbage collection time. The first column of the table shows the name of the compiler/interpreter (c.f. Table 3). The second column "route" duplicates the "route" column from Table 3. The third column gives a reference to the particular machine used (c.f. Table 13). The fourth column specifies whether the late ("L") or early ("E") formulation of the program gives the fastest execution. Columns 5 and 6 give the time and space required to execute the pseudoknot program. Unless noted otherwise, the space is the largest amount of space required by the program, as obtained from `ps -v` under the heading SIZE. The execution time of the C version of the pseudoknot program for the machine used may be found in Table 15. Please note that in most cases execution times and compilation times were measured on different machines. The last two columns "pseudoknots" show the relative and absolute performance with respect to C.

Again, broad performance groupings can be distinguished. The highest (relative) pseudoknots are measured for the Sisal compiler, which is actually faster than the corresponding C for the Cray version. The next best implementations are the Caml Gallium compiler (eager) and the Glasgow Haskell compiler (lazy). It is probably fair to say that the Glasgow Haskell version of the program has been more heavily optimised than the CAML Gallium version.

As the Clean and Glasgow Haskell compilers show, if the compiler can exploit strictness at the right points, the presence of lazy evaluation need not be a hindrance to high performance. It is also interesting that several of these compilers compile through C rather than being native compilers. Clearly, it is possible to compile efficient code without generating assembler directly. Space usage for these compilers is also generally low: the compilers have clearly optimised both for time and space.

The Lisp, Scheme and SML compilers generally yield very similar performance (35-50 pseudoknots). An obvious outlier is the Bigloo optimising Scheme compiler, whose performance is comparable to most

| compiler | route | mach. | time(s) | space | | C-time(s) | pseudoknots (%) | |
|---|---|---|---|---|---|---|---|---|
| | | | user + sys | Mb | [a] | user + sys | rel | abs |
| Gofer | I | 7 | 6.7 + 0.7 | 3 | A | 2.71 + 0.20 | 40.4 + 28.6 | 40.4 + 28.6 |
| RUFLI | I | 7 | 9.1 + 1.7 | 1 | A | 2.71 + 0.20 | 29.8 + 11.8 | 29.8 + 11.8 |
| Miranda | I | 7 | 12.5 + 0.8 | 13 | A | 2.71 + 0.20 | 21.7 + 25.0 | 21.7 + 25.0 |
| Caml Light | I | 2 | 29.7 + 1.1 | 2.3 | A | 2.66 + 0.05 | 9.0 + 4.5 | 9.1 + 18.2 |
| Clean | N | 4 | 30 + 10 | 9 | A | 2.69 + 0.55 | 9.0 + 5.5 | 9.0 + 2.0 |
| Trafola | I | 7 | 31.4 + 11.5 | 6 | A | 2.71 + 0.10 | 8.6 + 1.7 | 8.6 + 1.7 |
| RUFL | N | 7 | 41.6 + 8 | 3 | A | 2.71 + 0.20 | 6.5 + 2.5 | 6.5 + 2.5 |
| Poly/ML | I | 13 | 22.4 + 3.1 | 3.3 | A | 1.28 + 0.05 | 5.7 + 1.6 | 12.1 + 6.5 |
| Bigloo | C | 1 | 56.5 + 6.4 | 7.5 | A | 3.00 + 0.10 | 5.3 + 1.6 | 4.8 + 3.1 |
| Gambit | N | 8 | 147 + 77 | 14 | H | 6.34 + 0.12 | 4.3 + 0.2 | 1.8 + 0.3 |
| CMU CL | N | 5 | 118 + 25 | 14 | H | 4.73 + 0.43 | 4.0 + 1.7 | 2.3 + 0.8 |
| Camloo | S+C | 1 | 98 + 17.6 | 4.6 | A | 3.00 + 0.10 | 3.1 + 0.6 | 2.8 + 1.1 |
| Caml Gallium | N | 2 | 129 + 4.4 | 3.7 | A | 2.66 + 0.05 | 2.1 + 1.1 | 2.1 + 4.5 |
| SML/NJ | N | 16 | 131 + 4.4 | 50 | A | 1.90 + 0.10 | 1.5 + 2.3 | 2.1 + 4.5 |
| Facile | N | 20 | 123 + 2.5 | 11.3 | A | 1.71 + 0.08 | 1.4 + 3.2 | 2.2 + 8.0 |
| NHC(HBC) | I | 3 | 122 + 7.3 | 30 | A | 1.56 + 0.06 | 1.3 + 0.8 | 2.2 + 2.7 |
| Sisal (SUN) | N | 14 | 112 + 13.3 | 2.4 | A | 1.40 + 0.05 | 1.3 + 0.4 | 2.4 + 1.5 |
| MLWorks | N | 11 | 394 + 19 | 14.4 | R | 4.90 + 0.00 | 1.2 + 0.0 | 0.7 + 1.1 |
| Sisal (CRAY) | N | 15 | 82.5 + 15.5 | 24 | A | 0.76 + 0.02 | 0.9 + 0.2 | 3.3 + 1.3 |
| Yale | L | 5 | 610 + 186 | 14 | H | 4.73 + 0.43 | 0.8 + 0.2 | 0.4 + 0.1 |
| Chalmers | N | 3 | 181 + 45 | 50 | A | 1.33 + 0.06 | 0.7 + 0.1 | 1.5 + 0.4 |
| FAST | C | 7 | 450 + 40 | 100 | A | 2.71 + 0.20 | 0.6 + 0.5 | 0.6 + 0.5 |
| NHC(NHC) | I | 3 | 560 + 5.0 | 8.7 | A | 1.56 + 0.06 | 0.3 + 1.2 | 0.5 + 4.0 |
| Glasgow | C | 9 | 564 + 30 | 47 | A | 1.28 + 0.05 | 0.2 + 0.2 | 0.5 + 0.7 |
| Opal | C | 12 | 1301 + 19 | 15 | A | 3.00 + 0.08 | 0.2 + 0.4 | 0.2 + 1.1 |
| Erlang BEAM | C | 6 | > 1 Hour | 8 | A | 3.27 + 0.10 | — | — |
| CeML | C | 1 | > 1 Hour | 35 | A | 3.00 + 0.10 | — | — |
| ID | C | 10 | > 1 Hour | 64 | A | 2.75 + 0.05 | — | — |
| Stoffel | C | 17 | > 2 Hours | 25 | A | 1.28 + 0.05 | — | — |
| cc –O | N | 7 | 325 + 26 | 8 | A | 2.71 + 0.20 | 0.8 + 0.8 | 0.8 + 0.8 |
| gcc –O | N | 7 | 910 + 97 | 21 | A | 2.71 + 0.20 | 0.3 + 0.2 | 0.3 + 0.2 |

[a] A = Mbytes allocated space; H = Mbytes heap size; R = Mbytes maximum resident set size

Table 15: Results giving the amount of time (user+system time in seconds) and space (in Mbytes) required for compilation of the pseudoknot program. The "pseudoknots" give the relative speed with respect to the execution (not compilation) of the C version as a percentage.

| compiler | route | mach. | [a] | time(s) | space | | pseudoknots (%) | |
|---|---|---|---|---|---|---|---|---|
| | | | | user + sys | Mb | [b] | rel | abs |
| Sisal (CRAY) | C | 15 | L | 0.68 + 0.03 | 0.1 | A | 111 + 96.0 | 398 + 800 |
| Sisal (SUN) | C | 7 | L | 3.7 + 0.2 | 0.7 | A | 73.2 + 100 | 73.2 + 100 |
| Caml Gallium | N | 2 | L | 3.8 + 0.1 | 0.3 | A | 70.7 + 62.5 | 72.1 + 250 |
| Glasgow | C | 7 | L | 3.9 + 0.2 | 1 | A | 69.5 + 100 | 69.5 + 100 |
| Opal | C | 7 | L | 4.7 + 0.5 | 0.8 | A | 57.4 + 40.0 | 57.4 + 40.0 |
| Clean | N | 7 | L | 5.1 + 0.8 | 2.5 | A | 52.7 + 25.6 | 52.7 + 25.6 |
| CMU CL | N | 7 | L | 5.8 + 3.3 | 14 | H | 46.7 + 6.1 | 46.7 + 6.1 |
| Gambit | N | 8 | L | 17.3 + 0.2 | 14 | H | 36.6 + 60.0 | 15.7 + 100 |
| SML/NJ | N | 7 | L | 7.6 + 2.8 | 2.6 | A | 35.9 + 7.0 | 35.9 + 7.0 |
| MLWorks | N | 11 | L | 13.7 + 0.9 | 2.6 | A | 35.8 + 0.0 | 19.8 + 22.2 |
| CeML | C | 7 | L | 8.7 + 0.6 | 2 | A | 31.1 + 33.3 | 31.1 + 33.3 |
| FAST | C | 7 | L | 11.0 + 0.5 | 1 | A | 24.6 + 40.0 | 24.6 + 40.0 |
| Camloo | S+C | 7 | L | 11.5 + 1.3 | 4.9 | A | 23.6 + 15.4 | 23.6 + 15.4 |
| ID | C | 7 | L | 11.6 + 2.9 | 14 | A | 23.4 + 6.9 | 23.4 + 6.9 |
| Bigloo | C | 7 | L | 11.7 + 1.1 | 4.9 | A | 23.2 + 18.2 | 23.2 + 18.2 |
| Yale | L | 7 | L | 11.9 + 7.2 | 14 | H | 22.8 + 2.8 | 22.8 + 2.8 |
| Chalmers | N | 7 | L | 12.1 + 1.0 | 3 | A | 22.4 + 20.0 | 22.4 + 20.0 |
| Facile | N | 7 | L | 15.5 + 4.3 | 7.9 | A | 17.5 + 4.7 | 17.5 + 4.7 |
| Stoffel | C | 7 | L | 26.6 + 2.1 | 5.6 | A | 10.2 + 9.5 | 10.2 + 9.5 |
| Erlang BEAM | C | 7 | L | 31.8 + 4.5 | 11 | A | 8.5 + 4.4 | 8.5 + 4.4 |
| Caml Light | I | 7 | L | 53.9 + 7.5 | 0.3 | A | 5.0 + 2.7 | 5.0 + 2.7 |
| RUFL | N | 7 | L | 87 + 2.8 | 3 | A | 3.1 + 7.1 | 3.1 + 7.1 |
| Poly/ML | I | 13 | E | 54.7 + 8.4 | 3.3 | A | 2.3 + 0.6 | 5.0 + 2.4 |
| Trafola | I | 7 | L | 124 + 6.3 | 10.7 | A | 2.2 + 3.2 | 2.2 + 3.2 |
| NHC | I | 7 | E | 170 + 2.2 | 2.6 | A | 1.6 + 9.1 | 1.6 + 9.1 |
| Gofer | I | 7 | L | 370 + 12.0 | 3 | A | 0.7 + 1.7 | 0.7 + 1.7 |
| RUFLI | I | 7 | L | 529 + 13.0 | 4 | A | 0.5 + 1.5 | 0.5 + 1.5 |
| Miranda | I | 7 | L | 1156 + 34.0 | 13 | A | 0.2 + 0.6 | 0.2 + 0.6 |

[a] L = late formulation; E = early formulation
[b] A = Mbytes allocated space; H = Mbytes heap size

Table 16: Results giving the amount of time (user+system time in seconds) and space (in Mbytes) required for execution of the pseudoknot program. The "pseudoknots" give the relative speed with respect to the execution of the C version as a percentage.

of the non-strict implementations. The two SML compilers (SML/NJ and MLWorks) are particularly close in both time and heap usage.

Most of the non-strict compilers (Chalmers, FAST, Id, Stoffel, Yale and Glasgow Haskell on less optimised code) are grouped in the 10–25 pseudoknot range. These compilers typically offer around 75% of the performance of eager implementations such as SML/NJ or Gambit, or 50% of the performance of CMU Common Lisp. Even so, this level of performance has required the exploitation of strictness through unboxing and similar optimisations. Without these features, on the basis of the Glasgow results (Section 5.10), performance can be estimated as approximately 8 pseudoknots or just under a quarter of the typical performance of a compiler for an eager language. For these compilers and this application, laziness therefore costs directly a factor of 3, with a further 50% probably attributable to the use of different implementation techniques for predefined functions etc., which are required due to the possibility of laziness.

The lowest pseudoknot values are found with the interpretive systems, which are all less than 10 pseudoknots. This is not particularly surprising. The interpreters (Caml Light, Poly/ML, NHC, Trafola) which lie in the 1–10 pseudoknot range are, however, significantly faster than their conventional brethren, which are generally less than one pseudoknot. Interpreters for strict languages (CAML Light, Poly/ML, Trafola) do seem on the whole to be faster than interpreters for non-strict languages (NHC, Gofer, RUFLI, Miranda).

There are two remaining curiosities: the Erlang and RUFL compilers. These do not fit clearly in the same groups as other similar compilers. The low performance of the RUFL compiler may reflect its relative immaturity, as well as the fact that no special priority has been placed on floating-point performance.

The low performance recorded by the Erlang BEAM compiler reflects the fact that Erlang is a programming language primarily intended for designing robust, concurrent real time systems and a low priority has been placed on floating-point performance.

The set of CAML compilers offers an interesting spectrum: Caml Gallium is a slow compiler which produces fast code; Caml Light compiles quickly, but is relatively slow; and Camloo is intermediate between the two.

As might be expected, there is, broadly speaking, a strong relationship between compilation time and execution speed. Only the Clean implementation offers both fast compilation and fast execution. For the compiled systems there is also a very rough relationship between execution speed and heap usage: faster implementations use less heap. There does not, however, seem to be any correlation between non-strictness and heap usage.

The fact that Sisal is first-order may be significant, though this is hard to judge since the only other first-order implementation (Erlang) yields relatively poor performance. Sisal is also the only monomorphic language studied, so results here must also be inconclusive. Of the languages studied, Sisal is the only one that was specifically designed for "numeric" rather than "symbolic" computations, and clearly the design works well for this application. Floating-point performance has traditionally taken second-place in functional language implementations, so we may hope that these results spur other compiler writers to attempt to duplicate the Sisal results.

The provision of pattern-matching facilities does not appear to have any effect on code performance: there are fast and slow pattern-matching compilers. The fastest compilers all use strong type systems, which are known to assist compilation. The dynamically typed implementations are, however, intermediate in speed, and comparable with the statically typed SML systems.

# 7 Conclusions

Over 20 compilers for lazy and strict functional languages have been benchmarked using a single floating point intensive program. The C version of the program spends 25% of its time in the C library trigonometric and square root routines. The problem at hand may thus be typical only for geometric problems. The program should not be construed as a typical numerical scientific application. However, the program is useful as a benchmark, because the "real" work it does (the trigonometric and floating point calculations) is so clearly identifiable. Everything else the program does should be kept to a minimum. Not all implementations of functional languages that have been benchmarked are capable of realising this, but some are!

The compilation speed of most implementations (including the two C compilers) could be improved significantly. Generating C as intermediate code does not necessarily make the compiler slow, as demonstrated by the performance of the Bigloo and Camloo compilers, but generating fast C does often lead to high compilation times.

To achieve good performance from lazy implementations, it proved necessary to annotate certain data structures that are often used as strict. The performance of the pseudoknot program is not sensitive to incorrectly placed strictness annotations. In general, lazy functional programs are not so well behaved in this respect. Inserting annotations is a fine art, as demonstrated by the efforts of the Glasgow team. To make functional languages more useful than they are now, clearly more effort should go into providing users with simple to use and effective means of analysing and improving the performance of their programs.

Benchmarking a single program can lead to results which cannot easily be generalised. Special care has been taken to make the comparison as fair as possible: the pseudoknot program is not an essentially lazy program or an essentially non-lazy program; the different implementations use the same algorithm; most of the binaries were timed on one and the same machine.

Storing floating point numbers as unboxed values has a significant positive effect on the execution speed of the pseudoknot program. Most compilers can deal adequately with unboxed single precision floating point numbers. Few compilers can also implement unboxed double precision numbers properly. There is thus a need for better methods of dealing with double precision unboxed floats, although this problem might become obsolete with the next generation of 64 bit machine architectures.

There is no clear distinction between the runtime performance of eager and lazy implementations when appropriate annotations are used: lazy implementations have clearly come of age when it comes to implementing largely strict applications, such as the pseudoknot program. The speed of C can be approached by some implementations, but not without special measures such as strictness annotations. On the Cray, the Sisal version is faster than the C version.

## Acknowledgements

Mark Jones produced the Gofer version of the pseudoknot program.

Will Partain and Jim Mattson performed many of the experiments reported here for the Glasgow Haskell compiler. The work at Glasgow is supported by a SOED Research Fellowship from the Royal Society of Edinburgh, and by the EPSRC AQUA grant.

The work at Nijmegen is supported by STW (Stichting voor de Technische Wetenschappen, The Netherlands).

Jon-Dean Mountjoy performed some of the experiments for the RUFL implementation.

The ID version of the pseudoknot program was the result of a group effort. Credit goes to Jamey Hicks, R. Paul Johnson, Shail Aditya, Yonald Chery and Andy Shaw.

Zhong Shao made several important changes to the SML/NJ implementation, and Andrew Appel and David MacQueen provided general support in the use of this system. David Tarditi performed several of the SML/NJ experiments.

John T. Feo and Scott Denton of Lawrence Livermore National Laboratory collaborated on the Sisal version of the pseudoknot program.

## References

[1] M. Alt, C. Fecht, C. Ferdinand, and R. Wilhelm. The Trafola-S subsystem. In B. Hoffmann and B. Krieg-Brückner, editors, *Program development by specification and transformation, LNCS 680*, pages 539–576. Springer-Verlag, Berlin, May 1993.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge Univ. Press, Cambridge, England, 1992.

[3] J. Armstrong, M. Williams, and R. Virding. *Concurrent programming in Erlang*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[4] L. Augustsson. HBC user's manual. Programming Methodology Group Distributed with the HBC compiler, Depart. of Comp. Sci, Chalmers, S–412 96 Göteborg, Sweden, 1993.

[5] M. Beemster. The lazy functional intermediate language Stoffel. Technical report CS-92-16, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1992.

[6] M. Beemster. Optimizing transformations for a lazy functional language. In W.-J. Withagen, editor, *7th Computer systems*, pages 17–40, Eindhoven, The Netherlands, Nov 1993. Eindhoven Univ. of Technology.

[7] D. C. Cann. The optimizing SISAL compiler: version 12.0. Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, California, Apr 1992.

[8] D. C. Cann. Retire FORTRAN? a debate rekindled. *Communications ACM*, 35(8):81–89, Aug 1992.

[9] E. Chailloux. An efficient way of compiling ML to C. In P. Lee, editor, *ACM SIGPLAN Workshop on ML and its Applications*, pages 37–51, San Francisco, California, Jun 1992. School of Comp. Sci, Carnegie Mellon Univ., Pittsburg, Pennsylvania, Technical report CMU-CS-93-105.

[10] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *Programming Languages and System Architectures, LNCS 782*, pages 228–244, Zurich, Switzerland, Mar 1994. Springer-Verlag, Berlin.

[11] A. Diwan, D. Tarditi, and E. Moss. Memory subsystem performance of programs with copying garbage collection. In *21st Principles of programming languages*, pages 1–14, Portland, Oregon, Jan 1994. ACM.

[12] X. Leroy *et al. The Caml Light system, release 0.61*. Software and documentation distributed by anonymous FTP on `ftp.inria.fr`, 1993.

[13] M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Lisp and functional programming*, pages 119–130, Nice, France, Jul 1990. ACM.

[14] M. Feeley, M. Turcotte, and G. LaPalme. Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D structure determination. *Lisp and symbolic computation (to appear)*, 1994.

[15] S. Finn and M. Crawley. *Using Poly/ML 2.05M*. Abstract Hardware Ltd., Aug 1992.

[16] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Compiler construction*, pages 37–47, Montréal, Canada, Jun 1984. ACM SIGPLAN notices,19(6).

[17] R. Giegerich and R. J. M. Hughes. Functional programming in the real world. Dagstuhl seminar report 89, IBFI GmbH, Schloss Dagstuhl, D-66687 Wadern, Germany, May 1994.

[18] A. J. Gill and S. L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In *Proc. IFIP*, pages ???–???, Hamburg, Germany, Aug 1994.

[19] S. C. Goldstein. The implementation of a threaded abstract machine. Technical report UCB/CSP 94-818, Comp. Sci. Division (EECS), Univ. of California, Berkeley CA 94720, 1994.

[20] K. Gopinath and J. L. Hennesy. Copy elimination in functional languages. In *16th Principles of programming languages*, pages 303–314, Austin, Texas, Jan 1989. ACM.

[21] The Yale Haskell Group. *The Yale Haskell Users Manual (version Y2.1)*. Dept. of Comp. Sci, Yale Univ., Jul 1994.

[22] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on programming languages and systems*, 7(4):501–538, Oct 1985.

[23] Harlequin. *MLWorks draft documentation*. Harlequin Ltd, Cambridge, England, 1994.

[24] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. *Software—practice and experience*, 24(2):127–173, Feb 1994.

[25] P. H. Hartel and K. G. Langendoen. Benchmarking implementations of lazy functional languages. In *6th Functional programming languages and computer architecture*, pages 341–349, Copenhagen, Denmark, Jun 1993. ACM.

[26] B. Hausman. Turbo erlang: Approaching the speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, Boston/Dordrecht/London, Mar 1994.

[27] P. Hudak, S. L. Peyton Jones, and P. L. Wadler (editors). Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):R1–R162, May 1992.

[28] M. P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Dept. of Comp. Sci, Yale Univ., New haven, Connecticut, May 1994.

[29] B. W. Kernighan and D. W. Ritchie. *The C programming language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[30] X. Leroy. Unboxed objects and polymorphic typing. In *19th Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, Jan 1992. ACM Press.

[31] R. A. MacLachlan. CMU common Lisp user's manual. Technical report CMU-CS-92-161, School of Comp. Sci, Carnegie Mellon Univ., Jul 1992.

[32] J. R. McGraw, S. K. Skedzielewski, S. Allan, R. Oldehoeft, J. R. W. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language. Language reference manual version 1.2 M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, California, Mar 1985.

[33] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[34] R. S. Nikhil. ID version 90.0 reference manual. Computation Structures Group Memo 284-1, Laboratory for Comp. Sci, MIT, Cambridge Massachusetts, Sep 1990.

[35] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[36] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The glasgow Haskell compiler: a technical overview. In *Proc Joint Framework for Information Technology (JFIT) Conference*, pages ???–???, Keele, England, Mar 1993.

[37] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 636–666, Cambridge, Massachusetts, Sep 1991. Springer-Verlag, Berlin.

[38] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Concurrent Clean - version 1.0 - Language Reference Manual, draft version*. Dept. of Comp. Sci, Univ. of Nijmegen, The Netherlands, Jun 1994.

[39] J. E. Ranelletti. *Graph transformation algorithms for array memory memory optimization in applicative languages*. PhD thesis, Comp. Sci. Dept, Univ. of California at Davis, California, Nov 1987.

[40] J. A. Rees and W. Clinger. *Revised$^4$ Report on the Algorithmic Language Scheme*. MIT, Cambridge, Massachusetts, Nov 1991.

[41] N. Röjemo. Nhc - nearly a Haskell compiler. Technical report in preparation, Dept. of Computing Science, Chalmers Univ., 1994.

[42] P. M. Sansom and S. L. Peyton Jones. Generational garbage collection for Haskell. In *6th Functional programming languages and computer architecture*, pages 106–116, Copenhagen, Denmark, Jun 1993. ACM.

[43] W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Phoenix Seminar and Workshop on Declarative Programming*, pages 239–252, Sasbachwalden, West Germany, Nov 1991. Springer-Verlag, Berlin.

[44] M. Serrano. *Bigloo 1.7 user's manual*. INRIA Rocquencourt, France (to appear), 1994.

[45] M. Serrano and P. Weis. $1+1 = 1$: an optimizing Caml compiler. In *ACM-SIGPLAN Workshop on ML and its applications*, pages 101–111. Research report 2265, INRIA Rocquencourt, France, Nun 1994.

[46] Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton Univ, Princeton, New Jersey, Nov 1994.

[47] S. Smetsers, E. G. J. M. H. Nöcker, J. van Groningen, and M. J. Plasmeijer. Generating efficient code for lazy functional languages. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture, LNCS 523*, pages 592–617, Cambridge, Massachusetts, Sep 1991. Springer-Verlag, Berlin.

[48] G. L. Steele Jr. *Common Lisp the Language*. Digital Press, Bedford, second edition, 1990.

[49] B. Thomsen, L. Leth, S. Prasad, T.-S. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile antigua release – programming guide. Technical report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany (The reference manual and license agreement are available by anonymous ftp from ftp.ecrc.de.), 1993.

[50] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

[51] D. A. Turner. *Miranda system manual*. Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, Apr 1990.

[52] P. L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[53] P. Weis and X. Leroy. *Le langage Caml*. InterÉditions, 1993.

[54] E. P. Wentworth. Code generation for a lazy functional language. Technical report 91/19, Dept. of Comp. Sci, Rhodes Univ., Dec 1991.

[55] E. P. Wentworth. RUFL reference manual. Technical report 92/1, Dept. of Comp. Sci, Rhodes Univ., Jan 1992.