The Optimizing SISAL Compiler: Version 12.0

David C. Cann
Computing Research Group, L-306
Lawrence Livermore National Laboratory.
P.O. Box 808, Livermore, CA 94550
cann@sisal.llnl.gov

Abstract

This manual explains how to use version 12.0 of the Optimizing SISAL Compiler (OSC); exemplifies the execution of SISAL programs; and introduces and illustrates OSC's symbolic debugger (SDBX). Although not intended as a language tutorial, this manual will help the novice SISAL programmer get a feel for the language and learn how to use it effectively.

1 Introduction and Overview

This manual describes version 12.0 of the Optimizing SISAL Compiler (OSC); explains how to use it; exemplifies the execution of SISAL programs; and introduces and illustrates OSC's symbolic debugger (SDBX). This manual is a tutorial expansion of OSC's man-pages, and is loaded with examples and illustrations. We assume that the reader is familiar with SISAL [14], and has a basic understanding of parallel processing and the UNIX operating system[1].

We begin by introducing the SISAL project, presenting the motivation behind functional computing, and defining some terminology. Then we show the compilation and execution of a simple SISAL program, and provide an overview of OSC. Next, we describe the various compilation phases in more detail and discuss the most commonly used compile time and run time options, illustrating their functionality and importance. To complete the manual, we introduce OSC's mixed language interface, which allows SISAL programs to call C and FORTRAN, and FORTRAN and C programs to call SISAL; give hints and recommendations for improving program performance; identify the language features that are not supported; demonstrate OSC's symbolic debugger (SDBX); and provide instruction on where to obtain OSC and how to install it. The appendix provides a complete listing of the associated man-pages. Most of the examples in this manual are transcriptions of interactions with OSC and SISAL executables.

2 The SISAL Project

The SISAL project is a collaborative effort between Lawrence Livermore National Laboratory and Colorado State University. SISAL is a functional language for numeric computation [14].

[1] UNIX is a trademark of AT&T Bell Laboratories.

Functional languages promote the construction of correct parallel programs by isolating the programmer from the overwhelming complexities of parallel computing. SISAL in particular does not specify an execution model, nor does it assume the existence of special hardware. For example, it was the language of choice for the Manchester dataflow machine [8] and effectively exploits the Cray X-MP, Y-MP, and Cray 2 [4].

The primary goal of the SISAL project is to implement the language on both conventional and distributed memory multiprocessors [7]. Additional goals include the definition of a general-purpose functional language [16] and its intermediate forms [19,22]; the development of optimization techniques for high performance parallel functional computing [3,17]; the development of a microtasking environment that supports dataflow on conventional computer systems [5,18,23]; the realization of execution performance comparable to Fortran [4,6]; the validation of the functional style of programming for large-scale scientific applications [10]; and the creation of a self-sustaining user community to transfer the technology to industry [21,2].

3 Motivation for Functional Languages

Despite the commercial availability of multiprocessors, the number of parallel scientific and commercial applications in production use today remains virtually zero. For example, at the National Laboratories, most multiprocessor systems are still used as multiple single-processor machines, greatly reducing their computational power. Further, there is no demand for change, which ultimately stems from the lack of adequate parallel software in the market place. Put simply, the creation of correct, determinate parallel programs remains arduous, error prone, and costly. Here, three possible solutions have emerged:

1. Develop compilers that automatically parallelize imperative languages.
2. Extend imperative languages with constructs and primitives to express parallelism.
3. Develop new languages for parallel computing.

Even after extensive research and development, automatic parallelizing compilers for imperative languages have not met expectations. The fault is not entirely with the compilers. Developed for sequential machines, imperative languages are based on a model of computation that assumes a single processor and a single instruction stream. The model is not a natural candidate for parallel computing, and is not well suited for analysis. While small imperative programs may optimize well, larger, more complex codes quickly thwart effective optimization. Understanding the behavior of a large code requires global analysis, which is potentially of exponential complexity. If the compiler cannot determine if a potential dependency will prevent parallelization, it must assume the worst and preserve the dependency, consequently increasing synchronization and communication costs, and decreasing parallelization. Because of the many imperative codes in use today, these compilers will remain an alternative for some time to come, but will never represent an effective long term solution.

Extending imperative languages with constructs that allow the explicit expression of parallelism has proven difficult and error prone. The extensions often limit programmer productivity and hinder analysis. They fail to separate problem specification and implementation, hinder modular design, and inherently hide data dependencies. The expression of most parallel computations in these languages is verbose and unnatural. In addition to expressing the algorithm, the programmer must encode the program's synchronization and communication operations, ensure data integrity, and safeguard against race conditions. The extra programming complexity and the time-dependent errors exposed by this alternative can frustrate even the best of programmers [13]. Further, the lack of standardization has decreased portability across machines.

The third alternative, and the one we endorse, is the development of new parallel programming languages [1,9,11,12]. Functional languages such as SISAL expose implicit parallelism through data independence, and guarantee determinate results via their side-effect free semantics. A functional program comprises a set of mathematical expressions or mappings (the value of any expression only depends on the values of its inputs, and not on the order of their definition). The SISAL programmer simply expresses the operations constituting the algorithm. The scheduling of operations, communication of data values, and synchronization of concurrent operations are the responsibility of the compiler and run time system. The programmer does not and cannot manage these operations. SISAL programs that run correctly on a single processor are guaranteed to run correctly on any number of processors, regardless of architecture. Relieved of the most onerous chores of explicit parallel programming, the programmer is free to concentrate on algorithm design and application development.

4 Terminology

1. Product-form Loop: Sisal's parallel loop form (**for**).
2. Parallelization: Both the concurrentization and vectorization of product-form loops. For a concurrentized loop, blocks of consecutive iterations will execute on possibly different processors. All the iterations of a vectorized loop will execute on the same processor, but in a pipelined or SIMD fashion. A loop may be both concurrentized and vectorized.
3. Workers: The processors contributing to parallel execution.
4. Loop Slices: The number of iteration blocks created to realize the execution of a concurrentized loop. We refer to each block as a loop slice.
5. Parallel Nesting Level: The depth of a concurrentized loop in a nest of concurrentized loops. By convention, a parallel nesting depth of one refers to the outermost level of a nest.
6. Dynamic Storage Subsystem: The SISAL run time memory management facility.

5 Getting Started

In this section, we show the compilation and execution of a simple SISAL program. We assume the program, shown below, resides in file `example.sis`:

```
define main

function main( n:integer returns integer )
```

```
  1 + for i in 1,n returns value of sum 1 end for
end function
```

To compile the program, one simply types "`osc example.sis`", as transcribed below:

```
unix-> osc example.sis
 LL Parse, using binary files
* Reading file: example.sis...

version 1.8       (Mar 28, 1989)

  accepted
     5 lines in program
     0 errors ( calls to corrector)
     0 tokens inserted;    0 tokens deleted.
     0 semantic errors
unix->
```

The resulting executable is **s.out**, which by default takes its input from the standard input and writes its output to standard output, both in FIBRE format (see Section 14). If desired, however, both input-output can be redirected to files. For example,

```
s.out - outfile
```

writes the result to `outfile` (the dash represents standard input),

```
s.out infile
```

reads input from `infile`, and

```
s.out infile outfile
```

reads input from `infile` and writes results to `outfile`. Transcription of an execution follows:

```
unix-> s.out
CRAY SISAL 1.2 VERSION 12.0
4000000                      # ENTERED BY THE USER (FIBRE FORMAT)
4000001                      # PROGRAM OUTPUT (FIBRE FORMAT)
unix->
```

Note, `s.out` prints the version banner to standard error.

5.1 Renaming Executables

One can rename an executable by using **-o**. For example,

```
osc -o example.out example.sis
```

names the executable file `example.out`.

5.2 Program Entry Points

As demonstrated above, the default program entry point is function `main`. One can redefine it using **-e**. For example,

```
osc -e foo new_example.sis
```

uses function `foo`. A convenient alternative is to declare the entry point in the program text using an **entry** pragma having syntax

```
%$entry=name_list
```

where `name_list` represents a comma-separated list of entry point names. For example, `moo` is the entry point in

```
%$entry=moo
define moo

function moo( n:integer returns double_real )
  for i in 1,n returns value of sum 1.0D0 end for
end function
```

Note, two or more names can appear in an entry point name list if the corresponding functions are called by another language (see Section 16) or another SISAL module (see Section 13). A program can have only one operating system entry point.

6 Compiler Overview

In this section we present a overview of OSC and provide a brief introduction to its phases and subphases (see Figure 1). Later sections provide a more detailed look at the compiler and its optimizers.

The first phase of compilation translates SISAL source (taken to reside in files having suffix **.sis**) into an intermediate form know as IF1 [19]. By default, the source is run through the C preprocessor for file inclusion and macro expansion (see Section 7) Next, IF1LD links the generated IF1 files and forms a monolithic program or module that is then given to IF1OPT, IF2MEM and IF2UP for global optimization. IF1OPT is a machine-independent optimizer (see Section 8); IF2MEM is a build-in-place analyzer (see Section 9.1); and IF2UP is an update-in-place analyzer (see Section 9.2). The monolith, now optimized and translated into a second intermediate from, IF2 [22], is next given to IF2PART for parallelization (see Sections 10 and 11) and then to CGEN for C and FORTRAN code generation. The default is to generate C code only. The **-hybrid** option enables generation of both languages. On some machines, this can result in better code quality (especially on the Crays). OSC will invoke the local FORTRAN compiler to process the generated FORTRAN; however, it is the user's responsibility to provide the appropriate and desired compilation options, including optimization switches, etc. This can be done using the **-ff** option. For example,

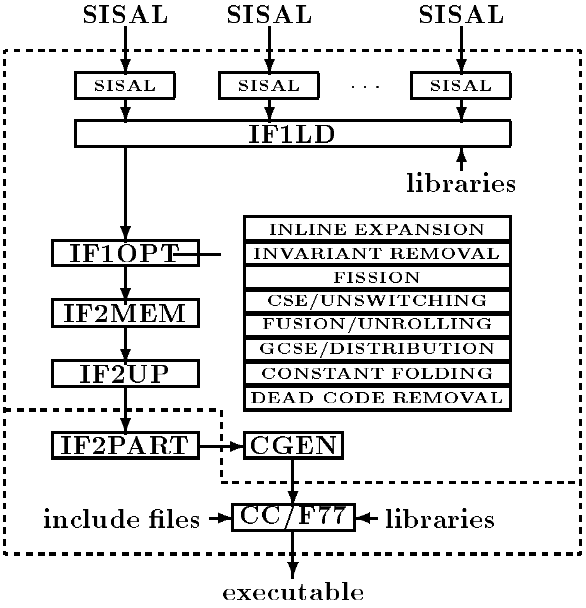

Figure 1: Internal Structure of OSC.

```
osc -hybrid -ff=-O example.sis
```

requests that OSC include "-O" on the FORTRAN compiler's command line. Further, one can use the **-FF** option to request use of a specific FORTRAN compiler:

```
osc -hybrid -FF=/usr/local/new_cf77 -ff=-Zv example.sis
```

OSC builds all its intermediate files in the current working directory. The intermediate files created for the compilation of `file1.sis` are

```
file1.i         % result of the C preprocessor
file1.if1       % result of SISAL
file1.lst       % cross reference tables and a listing given -listing
file1.mono      % result of IF1LD
file1.opt       % result of IF1OPT
file1.mem       % result of IF2MEM
file1.up        % result of IF2UP
file1.part      % result of IF2PART
file1.c         % result of CGEN
file1F.f        % result of CGEN given -hybrid
```

Unless told otherwise, OSC will remove all but the IF1 and FORTRAN files. Further, upon request, OSC will provide cross reference tables and a listing: **-listing**.

If desired, one can stop the compilation after file inclusion and macro expansion using **-CPP**; after front-end analysis and before IF1LD using **-IF1**; after IF1LD using **-MONO**; after CGEN using **-C**, and after assembly using **-noload**. One can restart compilation by providing the corresponding intermediate file(s) on the OSC command line. For example,

```
osc -CPP simple.sis
osc -IF1 simple.i
osc -MONO simple.if1
osc -C simple.mono
osc -noload simple.c
osc simple.o
```

and

```
osc simple.sis
```

are equivalent.

Note that the quality of the local C and FORTRAN compilers will determine the overall performance of a SISAL program. We chose C and FORTRAN as an intermediate form to shorten development time, increase system portability, and allow manual experimentation with future optimizations. We expect that better optimizing compilers for C will eliminate performance penalties incurred on some systems.

OSC does support separate compilation at the module level (see Section 13 for more details).

7 The C Preprocessor

By default, OSC will pass SISAL source files to the C preprocessor for file inclusion and macro expansion. One can disable this using **-nocpp**. OSC recognizes all the standard preprocessor options:

```
-Dname[=val]    % define macro name to have value val
-Uname          % undefine macro name
-Ipath          % establish path as one of the first directories searched
                % by the preprocessor during file inclusion
```

For example,

```
osc -DData_Size=200 -UDefault_Size -I/u0/cann/includes test.sis
```

defines macro `Data_Size` to have value `200`, undefines macro `Default_Size`, and establishes `/u0/cann/includes` as a search directory. The following SISAL program uses the C preprocessor commands to configure its compilation:

```
define main

#include "my_stdio.h"

#ifdef USE_DOUBLE
#define VALUE_TYPE double_real
```

```
#define VALUE 1.0D0
#else
#define VALUE_TYPE integer
#define VALUE 1
#endif

function main( n:integer returns VALUE_TYPE )
  for i in 1,n returns value of sum VALUE end for
end function
```

Warning: do not place commentary within macro definitions.

8 Scalar Optimizations: IF1OPT

By default, OSC runs with IF1OPT enabled. Applied optimizations include:

1. Function inlining.
2. Record and array fission.
3. Loop invariant removal.
4. Common subexpression elimination.
5. Inverse common subexpression elimination.
6. Dependent and independent loop and conditional fusion.
7. Loop unswitching.
8. Loop unrolling.
9. Loop distribution.
10. Constant folding and operator strength reduction.
11. Array stripping and dependence exposure.
12. Work reduction.
13. Dead code removal.

One can use the **-noscalar** option to disable everything but function inlining (see Section 8.13) and dead code removal (see Section 17.4).

In the remainder of this section, we illustrate the above optimizations and discuss function inlining and its control in more detail.

8.1 Record and Array Fission

Record and array fission are attempts to eliminate as many unnecessary record and array constructions as possible. For example,

```
let
  LoopRec := record LoopInfo [Lmin:1;Lmax:N;Kmin:1;Kmax:M;Shift:9];
in
  for I in LoopRec.Lmin,LoopRec.Lmax
  returns array of f( I+1+LoopRec.Shift );
  end for
end let
```

will become

```
for I in 1,N returns array of f(I+1+9) end for
```

eliminating the construction of `LoopRec`. The result is a more efficient program with more opportunities for further optimization. For example, the compiler can now fold `1+9` and form

```
for I in 1,N returns array of f(I+10) end for
```

8.2 Loop Invariant Removal

Loop invariant removal eliminates redundant computations across iterations, thus reducing loop execution time. For example,

```
V := for I in 2,n
       X := A[I]-A[1]*2.0;
     returns array of X
     end for;
```

will become

```
V := let
       T := A[1]*2.0;
     in
       for I in 1,N
         X := A[I]-T;
       returns array of X
       end for
     end let
```

When benchmarking, it is often useful to disable invariant removal from outermost loops. For an example, consider

```
define Main

type double = double_real;
type OneD   = array[double];

function Kernel1( n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
  for K in 1,n
    X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
  returns array of X
  end for
end function

function Main( rep,n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
  for i in 1, rep
    X := Kernel1( n, Q, R, T, Y, Z );
  returns value of X
  end for
end function
```

which benchmarks Kernel One of the Livermore Loops [15]. Note that the call to `Kernel1` is invariant. Hoisting this call, however, would preclude `rep` instantiations of the kernel. Thus for benchmarking, OSC supports an option called **-noOinvar** for disabling the removal of invariants from outermost loops.

8.3 Common Subexpression Elimination

Common subexpression elimination fuses identical computations within and between expressions, thus reducing the number of executed operations. For example,

```
V := R[I]*R[I];
```

will become

```
T := R[I];
V := T*T;
```

and

```
V := R[I]/2.0;
U := if ( B ) then R[I]/2.0 else R[I] end if
```

will become

```
T := R[I];
V := T/2.0;
U := if ( B ) then V else T end if
```

8.4 Inverse Common Subexpression Elimination

Inverse common subexpression elimination pushes identical operation sequences between branches (possibly having different operands) out of conditionals. For example,

```
V := if ( B[I] <= 0.0 ) then
       B[I]+C[I-1]/4.0
     else
       14.0+C[I-1]/C[I+1]
     end if;
```

will become

```
T1,T2 := if ( B[I] <= 0.0 ) then
           B[I],4.0
         else
           14.0,C[I+1]
         end if;

V := T1+C[I-1]/T3;
```

This optimization is experimental; hence, it is disabled by default. One can enable it using **-icse**. In general, we have observed that this optimization degrades run time performance.

8.5 Dependent and Independent Fusion

Fusion merges two or more compound computations into a single compound computation, thus reducing control overhead and exposing opportunities for further optimization. Independent fusion merges computations that have no data dependencies, and dependent fusion combines those with dependencies between them. Currently, IF1OPT considers only product-form loops having isomorphic generators and conditionals having isomorphic predicates. For example,

```
V1 := for X in A
        Y := exp(X,2.0);
      returns array of Y
      end for;

V2 := for X in A
        U := exp(X,3.0);
      returns array of U
      end for;
```

will become

```
V1, V2 := for X in A
            Y := exp(X,2.0);
```

```
          U := exp(X,3.0);
        returns array of Y
                array of U
        end for;
```

and

```
V1 := for X in A
        Y := X/2.0;
      returns array of Y
      end for;

V2 := for Z in V1
        U := Z+TEMP;
      returns array of U
      end for;
```

will become

```
V2 := for X in A
        Z := X/2.0;
        U := Z+TEMP;
      returns array of U
      end for;
```

Warning: in some situations independent loop fusion can reduce code quality; that is, overwhelm the local compilers. In general, this has only been observed on vector machines. In this regard, one can use **-nodfuse** and **-noifuse** to disable dependent and independent loop and conditional fusion respectively.

8.6 Loop Unswitching

Loop unswitching is an attempt to eliminate as many conditionals as possible from innermost loops, thus reducing the complexity of time critical computations. For example,

```
V := for I in 1,N
       Y := if ( I = 1 ) then 0.0 else A[I] end if;
     returns array of Y
     end for
```

will become

```
T := for I in 2,n
       Y := A[I];
     returns array of Y
     end for;
```

```
% THIS ASSUMES build-in-place analysis WILL BUILD T DIRECTLY INTO V
V := array_addl(T,0.0);
```

and

```
V := for I in 1,N
       Y := if ( B ) then 0.0 else A[I]*6.0 end if;
     returns array of Y
     end for
```

will become

```
V := if ( B ) then
       for I in 1,N returns array of 0.0 end for
     else
       for I in 1,N returns array of A[I]*6.0 end for
     end if;
```

8.7 Loop Distribution

See Section 11 for a complete discussion of loop distribution and options **-explode** and **-expodeI**. It suffices to say here that the intent of loop distribution is to expose additional opportunities for vectorization and concurrentization. Sometimes, this optimization can improve the effectiveness of loop fusion.

8.8 Loop Unrolling

IF1OPT will unroll product-form loops in their entirety if they can be predetermined to iterate `N` or fewer times (the default being 2 or less) or be within cost thresholds. This optimization helps reduce control overhead. One can override the default using **-u**. For example,

```
osc -u4 example.sis
```

will enable the unrolling of loops with four or fewer iterations, so

```
V := for I in 1,4
      X := foo(A[I])
     returns array of X
     end for;
```

will become

```
T1 := foo(A[1]);
T2 := foo(A[2]);
```

```
T3 := foo(A[3]);
T4 := foo(A[4]);
V  := array [1:T1,T2,T3,T4];
```

IF1OPT will not partially unroll a loop. This is left to the local compilers. Options **-nounroll** and **-u0** are equivalent and disable unrolling altogether.

8.9 Constant Folding and Operator Strength Reduction

IF1OPT does an aggressive job of constant folding and strength reduction. For example,

```
V := exp(N,10.0);
```

will become

```
T1 := N*N;
T2 := T1*T1*N;
V  := T2*T2
```

8.10 Array Stripping and Dependence Exposure

Array stripping is a form of copy elimination that identifies and eliminates unnecessary aggregate constructions. For example,

```
A := for I in 1,n
     returns array of Z[I]*Z[I]
     end for;

B := for I in 1,n returns array of A[I] end for;
```

will become

```
B := for I in 1,n
     returns array of Z[I]*Z[I]
     end for;
```

Array dependence exposure eliminates spurious dependencies; thus opening opportunities for further optimization. For example,

```
A := B[I:B[I]+B[I+1]];
C := A[J:A[I+1]+2];
```

will become

```
A := B[I:B[I]+B[I+1]];
C := A[J:B[I+1]+2];
```

allowing the elimination of one dereference operation:

```
T := B[I+1];
A := B[I:B[I]+T];
C := A[J:T+2];
```

8.11 Work Reduction

IF1OPT will attempt to eliminate unnecessary work by pushing branch specific expressions into referencing conditionals. For example,

```
A := B[I]*2.0;

V := if ( C[I-1] <= Z[J] ) then
       A*4.0
     else
       C[I+4]/Z[J+1]
     end if;
```

will become

```
V := if ( C[I-1] <= Z[J] ) then
       B[I]*8.0
     else
       C[I+4]/Z[J+1]
     end if;
```

8.12 Dead Code Elimination

IF1OPT will always do dead code elimination to avoid unnecessary computation. For example, consider

```
function goo( X:real returns real )
  let
    V,U := X*X*X, X/2.0;
  in
    10.0/V
  end let
end function
```

which will become

```
function goo( X:real returns real )
  let
    V := X*X*X;
  in
```

```
    10.0/V
  end let
end function
```

Sometimes it is possible to get some control over this optimization by using **-glue** (see Section 17.4).

8.13 Function Inlining

By default, OSC will attempt to inline as many non-recursive functions as possible. However, OSC will autamatically throttle the expansion so to prevent an explosion of code. One can use the **-inlineall** option to disable this throttle.

Like dead code elimination, **-noscalar** will not turn off function inlining. This must be done explicitly using **-noinline**. By using the **-iter** option, however, a programmer can gain interactive control over inlining. In this mode, IF1OPT will ask the user on a function by function basis if the functions are candidates for inlining. One can also use the **-call** option to disable inlining for specific function. For example,

```
osc -call sforme example.sis
```

will disable the inlining of `sforme`.

9 Copy Optimizations: IF1MEM and IF2UP

The most serious inefficiency in functional computing can be excessive storage management and data copying. OSC does an aggressive job of eliminating as much copy and management overhead as possible, in the form of build-in-place analysis (IF2MEM), update-in-place analysis (IF2UP), and array prebuilding: see [3,17].

9.1 Build-in-Place Analysis

Build-in-place analysis introduces run time code to preallocate array storage wherever possible [17]. For example, the final size of array `V` in

```
X := for I in 2,N returns array of I end for;
V := array_addl(X,1);
```

can be determined before the definition of `X`. Hence an `N` element vector can be preallocated and wired into the computation. Without this analysis, `N-2` intermediate arrays will be defined and discarded during the definition of `X`, and `X` will be copied into the storage allocated for `V`.

9.2 Update-in-Place Analysis

Update-in-place analysis reorders operations to allow as many write operations as possible to run in-place. For example, the execution order in

```
B,C := A[I:200],A[I];
```

is critical for optimization purposes. If the update executes first, it must modify a copy of A to preserve A for the selection. However, forcing the selection to execute first will allow the update operation to run in-place if there are no other accesses to A.

9.3 Array Prebuilding

By default, OSC will attempt to identify rectangular array constructions having invariant bounds and prebuild their storage frameworks. This helps to eliminate allocation-reclamation cycles in iterative computations. For example, without prebuilding

```
for initial
  I := 1;
  A := Ain;
while ( I <= N ) repeat
  I := old I + 1;
  A := for L in 1,K cross M in 1,K
       returns array of old A[L,M]+1
       end for;
returns value of A
end for
```

will build and reclaim N-1 instances of matrix A before returning its final value. This unoptimized translation resembles

```
for ( I = 1; I <= N; I++ ) {
  A = Malloc( K integer pointers );

  for ( L = 1; L <= K; L++ ) {
    R = Malloc( K integers );

    for ( M = 1; M <= K; M++ )
      R[M] = oldA[L][M]+1;

    A[L] = R;
    }

  for ( L = 1; L <= K; L++ )
    Free( oldA[L] );

  Free( oldA );

  oldA = A;
  }
```

With prebuilding, OSC, recognizing that A comprises K row vectors of size K and that K itself is loop invariant, will allocate and assemble the matrix before the loop executes:

```
/* PREBUILD THE FRAMEWORK FOR MATRIX A */
A = Malloc( K integer pointers );

for ( L = 1; L <= N; L++ ) {
  R = Malloc( K integers );
  A[L] = R;
  }

/* NOW DO THE ITERATIVE COMPUTATION */
for ( I = 1; I <= N; I++ ) {
  for ( L = 1; L <= N; L++ )
    for ( M = 1; M <= N; M++ )
      A[L][M] = oldA[L][M]+1;

  oldA = A;
  }
```

By default, OSC will only attempt to prebuild array aggregates having less than four dimensions. One can change this value using the **-pb** option. For example,

```
osc -pb4 example.sis
```

will allow arrays of up to four dimensions to be subjected to prebuilding. Note that `-pb0` disables the optimization altogether.

9.4 Summary

By default, all three optimizations are enabled, however **-nomem** will disable build-in-place analysis, **-noup** will disable update-in-place analysis, and **-nopreb** will disable array prebuilding (equivalent to **-pb0**). One can determine the success of copy elimination by using the **-copyinfo** option at compile time along with the **-r** option at run time (see Section 15.3). Note that specific styles of coding can hinder all three optimizations (see Section 17.1 for more information).

10 Automatic Concurrentization: IF2PART

On concurrent machines, the default is to run with concurrentization enabled. In general, the compiler does a good job of automatically identifying product-form loops that warrant concurrent execution[2]. One potential problem with the partitioner, however, is that it may under-parallelize a program; that is, it may underestimate loop iteration counts and computation costs. Fortunately, this rarely happens for large applications and the partitioning algorithm is easily tuned at compile time.

The first level of control is **-maxconcur**. This option causes IF2PART to parallelize product-form loops without regard to parallel nesting level or cost estimates. **Warning: this option**

[2] Function level parallelism is not currently supported.

should be used with caution, as it often results in over-concurrentization. The second level of control is **-noconcur**, which disables concurrentization altogether—this is the default on sequential machines. The third level of control tunes the partitioner via the command line. The parameters under command line control are:

1. `MAX_WORKERS`, which represents the maximum number of workers that could ever participate in concurrent execution. The default is the number of processors specified during OSC installation (see Section 20).
2. `MAX_PARALLEL_NESTING_LEVEL`, which defines the maximum parallel nesting level allowed for the program. The default is infinity (all levels).
3. `SLICE_THRESHOLD`, which defines the minimum execution cost estimate that enables loop concurrentized. The default is `7000` units.
4. `NUMBER_OF_ITERATIONS`, which defines the assumed iteration count for product-form loops having unknown iteration ranges. The default is `100` iterations.

The above parameters can be reset using the following options:

1. `MAX_WORKERS`: **-p** (example, `-p4`).
2. `MAX_PARALLEL_NESTING_LEVEL`: **-n** (example, `-n2`).
3. `SLICE_THRESHOLD`: **-h** (example, `-h8000`).
4. `NUMBER_OF_ITERATIONS`: **-i** (example, `-i1000`).

We present the partitioning algorithm below (`f` represents the product-form loop under consideration). In the current implementation, the value of AVG_VECTOR_SPEEDUP is fixed at 3.

```
TempCost         := f.BodyCost * NumberInterations(f);

ActualLoopCost   := if ( VectorCandidate(f) ) then
                      TempCost / AVG_VECTOR_SPEEDUP
                    else
                      TempCost
                    end if;

ConcurrentizeTheLoop,
WorkersThatAreNowBusy :=
  if ( ActualLoopCost >= SLICE_THRESHOLD &&
       CurrentParNestingLevel <= MAX_PARALLEL_NESTING_LEVEL &&
       old WorkersThatAreNowBusy < MAX_WORKERS ) then

    TRUE, if ( NumIterations(f) < MAX_WORKERS ) then
            old WorkersThatAreNowBusy * NumIterations
          else
```

```
            old WorkersThatAreNowBusy * MAX_WORKERS
          end if

    else

      FALSE, old WorkersThatAreNowBusy

    end if;
```

The value returned from `NumIterations` is either the actual number of iterations for function `f` or `NUMBER_OF_ITERATIONS`.

For implementation reasons, not all flavors of product-form loops are considered for concurrentization. OSC will not concurrentize the following four loop forms because of their return clauses:

```
A := for i in 1,n returns value of f(i) end for;

B := for i in 1,n returns array of i when g(i) end for;

C := for i in 1,n returns array of i unless g(i) end for;

D := for i in 1,n
       V := for j in 1,f(i) returns array of j end for;
     returns value of catenate V
     end for;
```

The last is not considered because the size of `V` is not invariant. However, OSC will consider a loop having the following returns clauses for concurrentization:

```
for I in 1,N
  X := A[I];
returns value of sum X
        value of product X
        value of greatest X
        value of least X
        value of catenate A
        array of X
end for
```

By default OSC considers reductions on real numbers associative, and therefore safe for concurrentization. If this is not desired, one can disable it using the **-noassoc** compile time option (see Section 11.4). Also, one can use **-cvinfo** to get compilation feedback about concurrentization (written to standard error).

11 Automatic Vectorization: IF2PART

As well as selecting product-form loops for concurrentization, IF2PART will automatically consider innermost product-form loops for vectorization. On vector machines, vectorization is the default. However, one can use **-novector** to disable it. The **-cvinfo** option will result in compilation feedback regarding vectorization. Note that OSC only makes recommendations for vectorization. That is, the local compilers must do the actual vectorization; OSC simply annotates the selected loops with vendor-specific pragmas.

In general, OSC selects almost all innermost loops for vectorization if they do not not create or manipulate records, unions, or streams, or make calls to other SISAL functions (not inlined by IF1OPT).

11.1 Vectorizable Sequential Loops

OSC will identify the following sequential loops as candidates for vectorization. Again, the final decision is made by the local compilers.

11.1.1 Tri-Diagonal Elimination

```
for initial
  K := 2; X := XIn[1];
while ( K <= N ) repeat
  X := Z[old K]*(Y[old K]-old X)
  K := old K + 1;
returns array of X
end for
```

11.1.2 Partial Sum

```
for initial
  K := 2; X := Yin[1];
while ( K <= N ) repeat
  X := old X + Yin[old K];
  K := old K + 1;
returns array of X
end for
```

11.1.3 Index of First Min or Max

```
for initial
  Idx := 1; K := 2;
while ( K <= N ) repeat
  % FIRST MIN
  % Idx := if ( X[old K] < X[old Idx] ) then
```

```
  % FIRST ABSOLUTE MIN
  % Idx := if ( abs(X[old K]) < abs(X[old Idx]) ) then

  % FIRST MAX
  % Idx := if ( X[old K] > X[old Idx] ) then

  % FIRST ABSOLUTE MAX
  Idx := if ( abs(X[old K]) > abs(X[old Idx]) ) then
           old K
         else
           old Idx
         end if;

  K := old K + 1;
returns value of Idx
end for
```

11.2 Loop Distribution

On a parallel machine, OSC will attempt to extract parallelism from sequential loops and vectorization from non-vector parallel loops. For example,

```
B := for initial
       I := 1;
       A := Ain;
     while ( I <= n ) repeat
       I := old I + 1;
       A := A[f(I):Z[old I]*W[old I]];
     returns value of A
     end for
```

will become

```
T := for I in 1,n
       v := Z[I]*W[I];
     returns array of v
     end for;

B := for initial
       I := 1;
       A := Ain;
     while ( I <= n ) repeat
       I := old I + 1;
       A := A[f(I):T[old I]];
     returns value of A
     end for
```

and

```
B := for I in 1,n
       A := Z[I]*W[I+1]+Y[I-1];
       V := for J in 1,n
            returns array of A*B[I,J]/0.456D0
            end for
     returns array of V
     end for
```

will become

```
T := for I in 1,n
       A := Z[I]*W[I+1]+Y[I-1];
     returns array of A
     end for;

B := for I in 1,n
       A := T[I];
       V := for J in 1,n
            returns array of A*B[I,J]/0.456D0
            end for
     returns array of V
     end for
```

In the interest of compile time, OSC is conservative in its loop distribution analysis. Two options exist to force a more aggressive and time consuming approach: **-explode** and **-explodeI**. The former has a global effect, while the later only considers innermost loops. For the previous example, **-explode** would produce:

```
T1 := for I in 1,n returns array of Z[I]*W[I+1] end for;

T2 := for I in 1,n returns array of T1[I]+Y[I-1] end for;

T3 := for I in 1,n cross J in 1,n
      returns array of T2[I]*B[I,J]
      end for;

B  := for I in 1,n cross J in 1,n
      returns array of T3[I]/0.456D0
      end for;
```

Because IF1OPT does the explosion, the transformed expressions are subject to further optimization (including loop fusion to intelligently repackage the loops).

11.3 Loop Fusion and Vectorization

When requesting vectorization, OSC will not fuse vector and non-vector loops, or innermost independent vector loops. This complements loop distribution.

11.4 Associativity

As with concurrentization, OSC will, by default, consider reductions on real numbers safe for vectorization. The **-noassoc** option will also disable such vectorizations.

12 Closing Comments about the Optimizers

There are several OSC options that subsume other options. First, **-noopt** implies **-noscalar**, **-nomem**, **-noup**, **-noconcur**, and **-novector** (but not **-noinline**). Second, both **-nomem** and **-noup** imply **-nopreb**. Third, **-seq** implies **-noconcur** and **-novector**. Fourth, **-info** implies **-cvinfo** as well as enabling IF1OPT, IF2MEM, IF2UP, and IF2GEN feedback. Fifth, **-nofuse** implies **-noifuse** and **-nodfuse**.

Options **-noup** and **-nomem** can devastate performance. A large problem may run 10 to 200 times longer without build-in-place and update-in-place analysis.

One can use the **-noimp** option to disable local compiler optimizations. Again, the user is responsible for configuring the optimization of generated FORTRAN code (**-hybrid**).

A program can be compiled for parallel execution but run on a sequential machine. Further, it is possible to cross-compile a SISAL program for the Crays using **-cray**. Here OSC will stop compilation after producing Cray specific C and FORTRAN code (that is, the option implies **-C**).

OSC has a verbose mode: **-v**. Here the compiler will echo to stderr each phase of compilation as it occurs. An example transcription follows:

```
unix-> osc -v -hybrid -ff=-O simple.sis
  /bin/sisal/spprun /bin/sisal/spp2 cc simple
  sisal -dir /bin/sisal simple.i -Fsimple.sis
  unlink simple.i
  if1ld -o simple.mono -FUR simple.if1
  if1opt simple.mono simple.opt -R -1 -l -e -U2
  unlink simple.mono
  if2mem simple.opt simple.mem
  unlink simple.opt
  if2up simple.mem simple.up -I
  unlink simple.mem
  if2part /bin/sisal/s.costs simple.up simple.part -P4
  unlink simple.up
  if2gen simple.part simple.c -KsimpleF -U -G -O -Y3
  unlink simple.part
  f77 -c -O simpleF.f
  cc -I/bin/sisal -DSGI -O -float -o s.out simple.c /bin/sisal/p-srt0.o
```

```
    -L/bin/sisal simpleF.o -lsisal -lmpc -lm
  unlink simple.c
```

13 Programs, Modules, and Separate Compilation

OSC supports the separate compilation of SISAL programs and modules to facilitate the development and maintenance of large applications. If one does not include **-module** on the OSC command line, then the provided files are linked to form a program. A program can have only one entry point which can only be called from the operating system. Two or more entry points are allowed, however, if they are called from another language (see Section 16). If **-module** is provided, then the files are linked to form a module. Here there can be one or more entry points, but they are only visible to other modules or a program.

The example we provided at the beginning of this manual (refer to Section 5) illustrated the compilation of a stand-alone program. Below, we illustrate use of the module facility. First we define a module comprising three functions, two of which are entry points. The first two functions reside in file1.sis:

```
%$entry=double
define double, mult

function mult( x,y:real returns real )
  x*y
end function

function double( x:real returns real )
  mult(x,x)
end function
```

The third function resides in file2.sis which imports function mult:

```
%$entry=triple
define triple

global mult( x,y:real returns real )

function triple( x:real returns real )
  mult(mult(x,x),x)
end function
```

The module (file1.o) is formed using

```
osc -module file1.sis file2.sis -noload
```

The third and final file (file3.sis) defines the program and references both the module's entry points:

```
define main

global double( x:real returns real )
global triple( x:real returns real )

function main( x:real returns real, real )
  double(x), triple(x)
end function
```

The program (file3.o) is formed using

```
osc file3.sis -noload
```

and the entire application (**s.out**) is formed using

```
osc file3.o file1.o
```

OSC supports a data base facility to improve the quality separately compiled code (**-db**). The user must provide a data base file name and include it on each OSC command line. For example, one could have formed the above application using data base `dbase`:

```
osc -module -db dbase file1.sis file2.sis -nolaod
osc -db dbase file3.sis -noload
osc -db dbase file3.o file1.o
```

The user need not be concerned with the contents of the data base, but it is human readable. When using this facility, we recommend that the modules be compiled bottom-up (that is, leaf modules up to the program, following the inter-module call chain).

14 Program Input-Output

As mentioned in Section 5, SISAL programs communicate with the outside world via FIBRE (see [20] for a comprehensive definition). In this section we illustrate the salient features of FIBRE using the program shown below. It takes as input a record, real, integer, boolean, vector, matrix, null, stream, string, and union:

```
define main

type Rec    = record[a:real;b:integer];
type String = array[character];
type OneI   = array[integer];
type OneD   = array[double_real];
type TwoD   = array[OneD];
type StrI   = stream[integer];
type Un     = union[a:real;b:OneD;c:null];
```

```
function main( a:Rec;     b:real; c:integer;
               d:boolean; e:OneI; f:TwoD;
               g:null;    h:StrI; i:String;
               j:Un
               returns integer )
  2010 % RETURN VALUE
end function
```

An example invocation follows.

```
unix-> s.out - output
SGI SISAL 1.2 V12.0
                                # BEGINNING OF FIBRE FORMATTED INPUT
< 2.0 1000 >                    # THE RECORD
4.123 66
T
[1: 1 2 3 4 5 6 7 8 9 10 ]      # THE VECTOR
[1:                             # THE MATRIX
  [1: 1.0d0 2.0d0 ]
  [1: 3.0d0 4.0d0 ]
]
nil                             # THE NULL
{ 3 2 1 }                       # THE STREAM
"hello world"                   # THE STRING
(1: [1: 1.0 2.0 3.0 4.0 5.0 ] ) # THE UNION (ORD(a)=0,ORD(b)=1)
                                # END OF FIBRE FORMATTED INPUT
unix-> more output
2010
unix->
```

As shown, the order and type of the provided inputs must match that declared in the function header of the entry point. The '#' characters begin FIBRE commentary, and are ignored during FIBRE processing.

To increase the performance of FIBRE processing, OSC supports the inclusion of upper bound specifications in array definitions. **We encourage the use of this enhancement.** Consider

```
[1,2:
   [2,5: 1 2 3 4 ]
   [2,6: 5 6 7 8 9 ]
]
```

which is equivalent to the older and slower form:

```
[1:
   [2: 1 2 3 4 ]
   [2: 5 6 7 8 9 ]
]
```

One can use the **-z** option to suppress program output altogether.

15 Program Execution

In this section we illustrate and discuss the various options and aspects of SISAL execution. These include options for configuring execution, tuning performance, and gathering performance data.

15.1 Parallel Execution

By default, a concurrentized program will only use one worker. One can use the **-w** option to realize parallel execution. For example,

```
s.out -w4
```

runs `s.out` with 4 workers, while

```
s.out -w1
```

and

```
s.out
```

use 1 worker.

During concurrent execution, the SISAL run time system splits concurrentized loops into `N` slices, where `N` is the number of participating workers. One can redefine `N` using **-ls**. For example,

```
s.out -ls100 -w4
```

will run `s.out` with 4 workers but with each concurrentized loop split 100 ways, not 4.

As an enhancement, the run time system does supports guided self scheduling, requested using the **-gss** option:

```
s.out -gss -w4
```

Here the slice thickness of a non-vectorized but compound concurrentized loop (that is, a loop that itself contains one or more loops) is dynamically computed as the number of iterations not yet scheduled divided by the number of participating workers. Under heavy system load, this option may increase program throughput. On an idle system, however, this option may degrade overall performance. Note that **-gss** and **-ls** conflict.

By default, loop slices, regardless of their means of generation, are randomly distributed to workers using a shared run queue. Using the **-b** option, the slices will be given to the workers in a consistent and regular manner (via a distributed run queue). This option can degrade performance if system load is high and gang scheduling is not supported within the operating system. This option will improve the exploitation of locality on cache coherent machines such as the Sequent Symmetry, however.

15.2 Memory Management

Each SISAL program runs with a fixed dynamic memory size. The default is 2 megabytes. If the memory is exhausted during execution, an error message is printed:

```
ERROR: (Alloc) ALLOCATION FAILURE: increase -ds value
```

The **-ds** option allows the user to define a new size. For example,

```
s.out -ds8000000
```

will run `s.out` with 8 megabytes of memory.

When a request is made for `Z` bytes of storage, the dynamic storage subsystem defines an exact fit to be a block of `S` bytes where `S >= Z` but `Z <= S+X`, where `X` is by default 100 bytes. One can use the **-dx** run time option to redefine `X`. For example,

```
s.out -ax200
```

sets `X` to 200 bytes. The option allows one to configure the storage subsystem to an application, but we have observed that 100 bytes is sufficient for most applications.

15.3 Performance Feedback

15.3.1 Execution Times

The **-r** option enables SISAL's run time performance monitor, which appends performance data to the resource file **s.info**. **Note: the data will not reflect the time required for FIBRE processing**. An example **s.info** file for a two worker execution follows:

```
 Workers   DsaSize  LpSliceV
       2  4000000b         2

    MemW      MemU
2624164b  2848408b

 CpuTime  WallTime    CpuUse
 35.5300   35.5300    100.0%
 35.5000   35.5300     99.9%
```

Note that timing data are given for each worker. `CpuTime` reflects the time a worker spent executing the program, while `WallTime` reflects the elapsed time between job submission and completion (a function of system load as well as computation time). `CpuUse` reflects the processor utilization for a worker (`CpuTime` divided by `WallTime`). The `Workers`, `DsaSize`, `LpSliceV` fields give the **-w**, **-ds** and **-ls** values used for the run. The `MemW` field gives the total number of bytes requested from the dynamic storage subsystem, and `MemU` gives the total number of bytes granted (the extra storage helps manage internal data structures).

One can request execution time summaries for individual function invocations using **-time**. For example,

```
osc -time conduc simple.sis
```

requests that timing data be collect for each invocation of function `conduc`. The run time system will print this data to stderr. This option prevents inline expansion of the target function.

15.3.2 Aggregate Copy Counts

Compiling a SISAL program with **-copyinfo** enables the gathering of aggregate copy information at run time (see Section 9). The data are written to **s.info** only if the **-r** is provided on the execution command line. An example follows:

```
 Workers   DsaSize  LpSliceV
       1  4000000b         1

    MemW      MemU
2621492b  2846168b

 CpuTime  WallTime    CpuUse
 65.7100   65.7100    100.0%

   AtOps  AtCopies     AcOps  AcCopies
 1567924         1     72416        97

   RcOps  RcCopies CharMoves
       0         0     77600
```

`AtOps` gives the total number of build-in-place attempts and `AtCopies` gives the total number of these attempts that failed to run in-place. Hence

```
1.0-(AtCopies/AtOps)
```

measures the success of build-in-place analysis for the run. `AcOps` gives the total number of update-in-place attempts, and `AcCopies` gives the total number of these attempts that failed to run in-place. Hence

```
1.0-(AcCopies/AcOps)
```

measures the success of update-in-place analysis for the run. `RcOps` and `RcCopies` are similar in meaning to `AcOps` and `AcCopies`, but concern records instead of arrays. The `CharMove` field gives the total number of bytes copied.

15.3.3 Floating Point Operation Counts

Compiling a SISAL program with **-flopinfo** enables the gathering of floating point operation counts at a cost to performance. The operations counted are `+`, `-`, `*`, `/`, and negation; reduction operations; logicals; and floating point intrinsics (sin, cos, etc.). Like the other performance gathering options, the counts are printed to the resource file. An example follows:

```
  Workers   DsaSize  ExactFit  DsaHelps
        1 10000000b      100b         0

     MemW      MemU  LpSliceV   ArrayEx
 3918552b  4252344b         1       100

  CpuTime  WallTime    CpuUse
  64.2100   64.2100    100.0%

 FlopCounts (ARITHMETIC):         119009291
               (LOGICAL):          19989885
             (INTRINSIC):           1904866
```

Note that the same program ran in 58 seconds without the counts.

One can get function by function summaries using the **-fflopinfo** option, which like **-time** disables inline expansion of the target function. Consider

```
osc -fflopinfo conduc -fflopinfo eos simple.sis
```

which request a count summary for both `conduc` and `eos`. Note that this option implies **-flopinfo**; however, the requested summaries are written to stderr and not the resource file.

16 Mixed Language Programming

In this section, we illustrate OSC's mixed language interface. The interface allows FORTRAN and C programs to call SISAL, and SISAL programs to call C and FORTRAN. Regardless of the direction, interface arguments and results types cannot involve streams, records, unions, nulls, and booleans. However, a routine called from SISAL can return more than one value.

16.1 Calling SISAL Kernels

Assume we want to call function `matrix` from FORTRAN:

```
%$entry=matrix
define matrix

type OneR   = array[real];
type TwoR   = array[OneR];
```

```
function matrix( AA:TwoR; l,k:integer returns TwoR, real )
  let
    RR,ss := for i in 1,l cross j in 1,k
             returns array of A[i,k]*2.0
                     value of sum A[i,k]
             end for
  in
    RR,ss
  end let
end function
```

The associated FORTRAN driver follows:

```
program mixed
real A(100,100), R(100,100)
real s
integer md(13)

C initialize matrix A (A FORTRAN SUBROUTINE!)
call minit( A, 100,100 )

C initialize the array descriptor for A and R

C column major, transpose the data, immutable
md(1) = 0
md(2) = 1
md(3) = 0

C description of the first dimension
md(4) = 1
md(5) = 100
md(6) = 1
md(7) = 100
md(8) = 1

C description of the second dimension
md(9)  = 1
md(10) = 100
md(11) = 1
md(12) = 100
md(13) = 1

C start the sisal run time system -w1, -ds4000000, -r
call sstart( 1, 4000000, 1 )

C call the sisal routine
call matrix( A,md, 100,100, R,md, s )
```

```
C shutdown the sisal run time system
call sstop

C print the results of execution, assuming mprint is a fortran
C subroutine that prints matrix R
print *, s
call mprint( R, 100,100 )

stop
end
```

Note the call to **sstart** to initialize the SISAL run time system, and the call to **sstop** to shut it down. Subroutine **sstart** takes three arguments. The first defines the number of desired workers (**-w**); the second defines the desired size of the dynamic storage subsystem (**-ds**); and the third enables or disables performance gathering (1 implies **-r**, 0 disables it). Subroutine **sstop** does not take any arguments. Both calls are required, and the SISAL run time system must be initialized before calling a SISAL function.

If desired, one can use subroutine **sconfig** to establish other SISAL run time options. This routine should be called before **sstart**. Subroutine **sconfig** takes five arguments. The first defines the loop slice value (**-ls**); the second enables guided self scheduling (**-gss**); the third establishes use of the distributed run queue (**-b**); the fourth sets the dynamic storage subsystem's exact fit threshold (**-dx**); and the last sets the array expansion threshold (**-ax**). For example,

```
call sconfig( -1, 1, 1, 300, 40 )
```

is equivalent to

```
s.out -gss -b -dx300 -ax40
```

respectively, and

```
call sconfig( 100, -1, 1, 300, 40 )
```

is equivalent to

```
s.out -ls100 -b -dx300 -ax40
```

respectively, and

```
call sconfig( -1, -1, -1, -1, -1 )
```

requests all the default values and is equivalent to not making the call.

An actual argument is required for each of the formal arguments to a SISAL call, and an argument is required for each of its results. The order of presentation must match that of the callee, and each array must be accompanied by an array descriptor (immediately following the array), which characterizes its structure and layout. In the above driver, for example, `A,md` maps to formal argument `AA`; `100,100` maps to formal arguments `l` and `k` respectively; `R,md` maps to `RR` (the first result); and `s` maps to `ss` (the second result). Actual arguments `R` and `s` provide the target storage.

The most complicated aspect of an interface call is the array descriptor. The first component of a descriptor defines the majority aspect of the associated array, be it **column** major (encoded by `0`) or **row** major (encoded by `1`). FORTRAN implementations use column major, while C uses row major. The second component defines the flavor of data movement across the interface. A value of 1 (as in the example above) requests data transposition. Hence, for matrix `A` in the example, `A(2,1)` in FORTRAN will become `A[1,2]` in SISAL and `R[2,1]` in SISAL will become `R(1,2)` in FORTRAN. A value of `0` would preserve the original ordering. The third component specifies the mutability of the transmitted data, where `1` represents mutable and `0` represents immutable. A mutable array may have its contents changed within the called routine. When the descriptor is associated with a result, the third component is meaningless and ignored by the interface.

The remaining components of a descriptor occur in groups of five, one for each dimension of the array (in increasing dimension order). The first component of a group defines the associated dimension's actual lower bound. The second component defines the associated dimension's actual upper bound. The third and fourth components define the desired logical lower and upper bounds of the dimension. These identify the index set of the data to be moved across the interface. The meaning of the last component depends on the descriptor's context of use. If the descriptor is describing an argument, the last component defines the desired lower bound of the dimension once transmitted into SISAL. If the descriptor is describing a result, the last component defines the index of the first value to be returned from the dimension. Note that the above example transmits `A` and `R` in their entirety.

To further illustrate the functionality of an array descriptor consider the following FORTRAN code fragment, where Z and X are six-element integer arrays and `md` is a descriptor:

```
C INITIALIZE Z AND X
Z(1) = 1
Z(2) = 2
Z(3) = 3
Z(4) = 4
Z(5) = 5
Z(6) = 6

X(1) = 0
X(2) = 0
X(3) = 0
X(4) = 0
X(5) = 0
X(6) = 0

C INITIALIZE THE DESCRIPTOR
```

```
C COLUMN MAJOR, PRESERVE THE INDEX SET, IMMUTABLE
md(1) = 0
C md(2)=1 IS LEGAL, BUT IGNORED WHEN DESCRIBING A VECTOR
md(2) = 0
md(3) = 0

C SIX ELEMENT ARRAY, TRANSFER ONLY 3 OF THE ELEMENTS, 7 IS
C THE SISAL LOWER BOUND.
md(4) = 1
md(5) = 6
md(6) = 3
md(7) = 5
md(8) = 7
```

Now passing `Z` to

```
function foo( ZZ:OneI returns integer )
  ZZ[7]+ZZ[8]+ZZ[9]
end function
```

using

```
iresult = foo( Z,md )
```

yields `3+4+5`, while passing `X` to

```
function goo( returns ZZ:OneI )
  array [5: 10, 20, 30, 40, 50, 60, 70 ]
end function
```

using

```
call goo( X,md )
```

yields

```
X(1) equal to 0
X(2) equal to 0
X(3) equal to 30
X(4) equal to 40
X(5) equal to 50
X(6) equal to 0
```

Note that the following example descriptor describes an empty one dimensional transmission because the logical upper bound is one less than the logical lower bound:

```
md(1) = 0
md(2) = 0
md(3) = 0
md(4) = 1
md(5) = 6
md(6) = 1
md(7) = 0
md(8) = 1
```

When compiling a program that calls SISAL, one must provide either **-forFORTRAN** or **-forC** on the command line, depending on the language of the driver:

```
osc -forFORTRAN driver.f matrix.sis
```

The C version of program `mixed` follows. Note that the arguments are passed by reference:

```
main()
{
  float *A,*R,s;
  integer md[13], procs, ds, sinfo;
  integer l,k;

  l = k = 100;

  A = (float *) malloc( l*k*sizeof(float) );
  R = (float *) malloc( l*k*sizeof(float) );

  minit( A, 100,100 );

  md[0] = 1;
  md[1] = 1;
  md[2] = 0;

  md[3] = 0;
  md[4] = 99;
  md[5] = 0;
  md[6] = 99;
  md[7] = 1;

  md[8]  = 0;
  md[9]  = 99;
  md[10] = 0;
  md[11] = 99;
  md[12] = 1;

  procs = 1; ds = 4000000; sinfo = 1;

  sstart( &procs, &ds, &sinfo );
```

```
    matrix( A,md, &l,&k, R,md, &s );
    sstop();

    printf( "%e\n", s );
    mprint( R, 100,100 );
  }
```

For C, arguments to **sconfig** must also be passed by reference:

```
  lsv = 100; gss = -1; b = 1; xft = 300; ax = 40;

  sconfig( &lsv, &gss, &b, &xft, &ax );
```

Note that a program can specify two or more entry points if they are called by another language:

```
  %$entry=matrix,vec,foo,goo
```

In general, the mixed language interface can be expensive, especially for repetitive invocations:

```
     do 10 i = 1,100
       call sentry( A,md, R,md )
       A(1,i) = i
  10 continue
```

However in this example, note that both the descriptors (addresses and contents) and array argument addresses do not change between successive invocations. If this is the case for all calls into SISAL, then compiling the SISAL program with **-bind** may dramatically reduce the cost of the interface. The following excerpt is not a candidate for binding because the first argument address may change between invocations:

```
     do 10 i = 1,100
       if ( i .le. 50 ) then
         call sentry( A,md, R,md )
       else
         call sentry( B,md, R,md )
       endif
  10 continue
```

16.2 Calling Non-SISAL Kernels

In addition to allowing FORTRAN and C programs to call SISAL, SISAL functions can call FORTRAN and C. The SISAL code must provide a global declaration for each of the called routines and identify the respective languages. For example, function `main` shown below calls function `halve` written in FORTRAN (as specified by a **fortran** pragma, which has syntax similar to the **entry** pragma):

```
%$fortran=halve
define main

global halve( x:real returns real )

function main( x:real returns real )
  halve(x)
end function
```

The associated function template for `halve` follows:

```
function real halve( x )
real x
halve = .....
return
end
```

The C version of the above example requires

```
%$c=halve
```

instead of

```
%$fortran=halve
```

with the function template redefined as

```
float halve( x )
float x;
{
  return( ..... );
}
```

A more complicated example is shown below. Function `ftemp`, written in FORTRAN, returns an array. The last actual argument to this routine defines the result's descriptor, which controls the transmission. Note that unlike calls to SISAL, the array arguments do not need descriptors; they are always transmitted in their entirety and without transposition.

```
%$fortran=ftemp
define stemp

type OneR = array[real];
type OneI = array[integer];

global ftemp( u,v:OneR; k:integer; vd:OneI returns OneR );
```

```
function stemp( u,v:OneR; k:integer returns OneR )
  let
    vd := array [1: 0,0,0, 1,k,1,k,1];
  in
    ftemp( u,v,k, vd )
  end let
end function
```

The definition of `ftemp` follows. Because it returns an array, `ftemp` is a subroutine and not a function. The fourth argument points to the result's array descriptor (`vd`), and the fifth argument points to the storage for the result (`r`).

```
    subroutine ftemp( u,v,k, vd, r )
    real u(*),v(*),r(*)
    integer k, vd(*)

    do 100 i = 1,k
      r(i) = u(i)*v(i)
100 continue

    return
    end
```

An even more complicated example follows. Here function `main` calls function `big`, written in FORTRAN, which takes a scalar as input and returns two scalars as output:

```
%$fortran=big
define main

global big( x:real returns integer, real )

function main( x:real returns integer, real )
  big(x)
end function
```

The associated template for `big` follows. The template defines a subroutine because the function returns more than one value:

```
subroutine big( x, r1, r2 )
real x, r2
integer r1
r1 = ...
r2 = ...
return
end
```

Now we exemplify a FORTRAN routine that takes an array and scalar as input and returns two scalars and and two arrays as output (the second array being two dimensional). Note that the last two arguments define the results' descriptors:

```
%$fortran=biggest
define main

type OneR = array[real];
type TwoR = array[OneR];
type OneI = array[integer];

global biggest( u:OneR; k:integer; vd1,vd2:OneI
                returns real, OneR, real, TwoR );

function main( u:OneR; k:integer returns real, OneR, real, TwoR  )
  let
    vd1 := array [1: 0,0,0, 1,k,1,k,1];
    md1 := array [1: 0,0,0, 1,4,1,4,1,  1,4,1,4,1];
  in
    biggest( u,2010, vd1,md1 )
  end let
end function
```

The associated template follows:

```
subroutine biggest( u,k, vd1,md1, r1, rv1, r2, rm1 )
real u(*),rv1(*),rm1(4,4)
integer k, vd1(*),vd2(*)

r1 = ...
r2 = ...

rv1(1) = ...

rm1(1,1) = ...

return
end
```

In general, one can use the **-f** and **-c** compile time options instead of pragmas to associate languages with global routines. For example,

```
osc -f halve example.sis
```

is equivalent to

```
%$fortran=halve
```

and

```
osc -c halve example.sis
```

is equivalent to

```
%$c=halve
```

17 Warnings, Hints, and Recommendations

In this section we give hints and recommendations for better performance, enumerate unimplemented language features, and discuss approaches for debugging SISAL programs.

17.1 Things to Avoid

The OSC optimizers are powerful, but are not a panacea. As a general rule, write FORTRAN style SISAL to get the best optimized performance. However in the interest of parallelism and ease of expression, use product-form loops wherever possible. Remember, to determine the potential cost of data copying, use the **-copyinfo** and **-info** options in the early stages of program development (see Sections 15.3 and 12 respectively). Further, if possible, adhere to the following coding rules to get the best scalar, vector, and concurrent performance:

1. Avoid incremental constructions that extend arrays contained in other arrays or extend arrays built in other contexts. For example,

   ```
   V := A[i] || B[i]
   ```

   and

   ```
   type OneR = array[real];

   function foo( A:OneR; N:integer; returns OneR )
     let
       newA := array_addh(A,0.0);
     in
       if ( N = 1 ) then
         newA
       else
         foo(newA,N-1)
       end if
     end let
   end function
   ```

   will both result in data copying. The former may be unavoidable, however the later is better expressed as

   ```
   array_fill(1,N,0.0)
   ```

   Note that appending a value to a row of a matrix may result in copying:

```
V := array_addh(old A[i],0.0)
```

In general, this depends on the overall computation. For example, if V defines a replacement for A[i] and is the last use of A[i], then it should run in-place:

```
V := array_addh(old A[i],0.0)
A := old A[i:V];
```

2. Avoid loops whose iteration counts cannot be easily precalculated before execution, such as

```
for I in 1,N
  X := foo(I);
returns array of X when f(I)
end for
```

and

```
for initial
  I,V := 1,1;
while ( I <= N ) repeat
  I := old I*V;
  V := f(I)
returns array of V
end for;
```

and

```
V := for B in A
       Z := for v in B
              returns array of f(v)
              end for;
     returns value of catenate Z
     end for;
```

Each expression incrementally builds successively larger arrays, copying the smaller into the larger. However, the SISAL run time system will try to reduce this copying. Each time extra storage is required to hold an array, the system will allocate additional storage in anticipation of further expansions; that is, storage for N extra elements will be allocated, where N is NUM times the number of previous expansions for the array. By default, NUM is 100. One can use the **-ax** run time option to change this value. For example,

```
s.out -ax200
```

changes NUM to 200.

3. Avoid the use of **array_adjust** in computations such as

```
A := for j in 0,N returns array of double_real(j) end for;
L := array_adjust(1,M,A);
R := for i in M+1,N returns array of A[i]*2.0 end for;
V := L || R;
```

A better approach is to build `A` to the desired size:

```
A := for j in 1,M returns array of double_real(j) end for;
L := M;
R := for i in M+1,N returns array of A[i]*2.0 end for;
V := L || R;
```

4. Avoid cyclic computations that build aggregates whose sizes vary over time:

```
for initial
  I := 1; V := 0.0;
while ( I <= N ) repeat
  I := old I + 1;
  M := for J in 1,old I
       returns array of J        % OF SIZE old I
       end for;
  X := M[old I];
returns value of X
end for
```

This type of computation may fragment memory in the dynamic storage subsystem, hence increase management overhead. For this example, the best performance may be had by building arrays of similar size and relying on parallel execution to hide the unnecessary costs:

```
for initial
  I := 1; V := 0.0;
while ( I <= N ) repeat
  I := old I + 1;
  M := for J in 1,N
       returns array of J        % ALWAYS OF SIZE N
       end for;
  X := M[old I];
returns value of X
end for
```

5. Avoid expressions that introduce spurious dependencies, such as

```
for initial
  I := 1; V := Ain;
while ( I < N ) repeat
  I := old I + 1;
  V := old V[I:Ain[I+1]];
returns value of V
end for
```

which introduces copying to preserve `Ain`. A better expression would replace the fifth line with

```
V := old V[I:old V[I+1]];
```

Even better, the entire expression can be written using a product-form loop:

```
for I in 1,N
  X := if ( I = N ) then Ain[I] else Ain[I+1] end if;
returns array of X
end for
```

6. Avoid column major computations, as SISAL is row major. For example,

```
for i in 1,M cross j in 1,N
returns value of sum A[j,i]
end for
```

will run slower than

```
for j in 1,N cross i in 1,M
returns value of sum A[j,i]
end for
```

7. Avoid the use of implicit iteration bounds wherever possible. For example, if the lower bound of array `A` is known to be `world.Lmin` and the upper bound is known to be `world.Lmax` then use

```
for I in world.Lmin,world.Lmax
  V := A[I];
returns ...
end for
```

to scatter the components instead of

```
for V in A
returns ...
end for
```

8. Avoid the use of records or unions as array constituents. However, OSC will do a reasonable job of optimizing computations using records to define simple data such as complex values:

```
type Complex = record[ r,i:real ];

type OneC = array[Complex];          % THIS IS OK!
type TwoC = array[OneC];             % THIS IS OK!
```

9. To increase the success of prebuilding, avoid aggregate sharing wherever feasible (see Section 9.3). For example, instead of

```
newAC := for I in 2,N-1 cross J in 2,N-1
            V := old A[I,J]*3.0+projct(Z[I,J+1],Z[I,J-1],Z[I,J]);
          returns array of V
          end for;

A1    := array_addl( newAC, old A[1] );
A     := array_addl(    A1, old A[N] );
```

use

```
% THE DEFINITION OF newAC STAYS THE SAME, BUT USE
A1    := array_addl( newAC, acopy(1,N,old A[1]) );
A     := array_addl( A1,    acopy(1,N,old A[N]) );
```

where function acopy makes an explicit copy of the specified row:

```
function acopy( lo,hi:integer; R:array[real] returns array[real] )
  for I in lo,hi
  returns array of A[I]
  end for
end function
```

Array prebuilding requires the complete reconstruction of the target array. In general, the overhead of copying will be minimal compared to the expense of not prebuilding the array, especially on the Crays where the data movements vectorize.

10. Avoid interface calls to SISAL routines that are not computationally intensive. In general the interface is expensive, especially when passing and returning multidimensional arrays.

17.2 Some Things Not to Avoid

Below we list four example expressions that will run in-place after copy optimization. The first expression illustrates an efficient use of **returns value of catenate**:

```
% A IS ASSUMED TO BE RECTANGULAR!
Lm := array_liml(A);
Lx := array_limh(A);
Km := array_liml(A[Lm]);
Kx := array_limh(A[Lm]);

% NOTE:
%  THE SIZE OF Z IS INVARIANT: Kx-Km+1
%  THE SIZE OF V IS INVARIANT TO ITS CONSTRUCTION: (Lx-Lm+1)*(Kx-Km+1)

V := for L in Lm,Lx
       B := A[L];
       Z := for K in Km,Kx
              v := B[K];
```

```
           returns array of f(v)
           end for;
    returns value of catenate Z
    end for;
```

The second expression illustrates an effective use of `array_addh` and `array_addl` when building a matrix having a border of zeroes:

```
Zeroes := array_fill(1,N,0.0);

% NOTE:
%   THE SIZE OF M is Lx-Lm+1, WHICH IS KNOWN BEFORE C IS DEFINED

C := for L in Lm+1,Lx-1

       % NOTE:
       %   THE SIZE OF CR IS kx-km+1, WHICH IS KNOW BEFORE CRC IS
       %   DEFINED.

       CRC := for K in Km+1,Kx-1
              returns array of f(K)
              end for;

       CR  := array_addl(array_addh(CRC,0.0),0.0);
     returns array of CR
     end for;

M := array_addl(array_addh(C,Zeroes),Zeroes);
```

The third expression illustrates an effective incremental construction using `array_addl`:

```
% NOTE:
%   THE FINAL SIZE OF V IS N, WHICH IS KNOWN BEFORE THE INITIAL
%   DEFINITION OF A

V := for initial
       A := array OneI [];
       I := 1;
     while ( I <= N ) repeat
       I := old I + 1;
       A := array_addl(old A,old I);
     returns value of A
     end for
```

The fourth expression illustrates an effective way to mask out implicit first iterations:

```
for initial
```

```
  I, V,First := 1,0.0,TRUE;
while ( I <= N ) repeat
  I := old I + 1;
  V := foo( old I );
  First := FALSE;
returns array of V unless First
end for
```

17.3 Unimplemented Language Features

OSC does not support the following features of SISAL and FIBRE:

1. Array component repetition in FIBRE is not supported.
2. `stream_size` and `stream_prefixsize` are not supported.
3. The **returns old** clause is not supported.
4. Only two features of SISAL 1.2's error semantics are supported: `is error` and **error constants**. However, `is error` execution always yields **false**, and **error constant** generation always terminates execution. For example,

   ```
   define main

   function main( returns integer )
     error[integer]
   end function
   ```

   will yield

   ```
   ERROR: (example.sis,main,line=3) EXPLICIT ERROR VALUE GENERATED!
   ```

   Note that all other situations that should yield error values will yield unpredictable execution; some may cause program termination and some may not.
5. Reduction orders **left**, **right**, and **tree** are interpreted as **left** unless part of a parallelized loop where they are interpreted as random (see Section 11.4).
6. OSC implements streams strictly, so computations defining large streams will exhaust the dynamic storage subsystem and result in program termination.

17.4 Debugging

Debugging a SISAL program can be trivial depending on the size of the application and the degree of abstraction and hierarchical design. We recommend that large applications be debugged in small pieces, testing the lowest levels first, and then working up the call tree.

It has been our experience that most bugs result from illegal array references, themselves resulting from confusion over lower bound assignments in loop computations. The following expression should help clear the confusion; it yields 2, 1, 1, and 1 respectively:

```
function test( returns integer, integer, integer, integer )
  let
    % NOTE: a WILL HAVE A LOWER BOUND OF 2
    a := for i in 2,3 returns array of i end for;

    % NOTE: b WILL HAVE A LOWER BOUND OF 1, NOT 4!!!
    b := for i in 4,6 returns value of catenate a end for;

    % NOTE: c WILL HAVE A LOWER BOUND OF 1
    c := for initial
           i := 2;
         while ( i < 3 ) repeat
           i := old i + 1;
         returns array of i
         end for;

    % NOTE: d WILL HAVE A LOWER BOUND OF 1
    d := for initial
           i := 2;
         while ( i < 3 ) repeat
           i := old i + 1;
         returns value of catenate c
         end for;
  in
    array_liml(a),array_liml(b),
    array_liml(c),array_liml(d)
  end let
end function
```

In summary, an array defined by a `for initial` loop using `array of` or `value of catenate` will have a lower bound of 1, as will an array defined by a product-form loop using `value of catenate`. The lower bound of an array defined by a product-form loop using `array of` will have the value of the loop's first index.

To isolate an illegal dereference, OSC supports a compile time option called **-bounds** that enables run time bounds checking at a cost to performance. For example,

```
define main

function main( returns real )
  let
    arr := array[1: 1.0, 2.0 ];
    idx := array_limh(arr)+1;
    V   := arr[idx];
  in
    V
  end let
end function
```

yields

```
TOKEN: (arr,lo=1,size=2)
TOKEN: (idx,val=3)

ERROR: (example.sis,main,line=7) ARRAY SUBSCRIPT VIOLATION [HIGH]
```

if it is compiled with **-bounds**. From the run time message we see that the dereference on line 7 is attempting to select the third element of an array have only two elements. **Note that bounds checking disables vectorization.**

To examine intermediate results, OSC supports a predefined function called **peek** that accepts zero or more arguments of any type and returns integer 1. The function prints its input values to standard error in FIBRE format. For example,

```
V := 100;
DUMMY1 := peek(V);
U := 200;
DUMMY2 := peek(DUMMY1,U);
```

will print `100`, 1, and `200` to stderr respectively. Note that the reference to DUMMY1 in the second call enforces the print order; that is,

```
V := 100;
DUMMY1 := peek(V);
U := 200;
DUMMY2 := peek(1,U);
```

may print 1 and `200` before `100` (here the programmer is at the mercy of OSC). Regardless, OSC will at no time remove dead **peek** code.

If desired, one can also use the mixed language interface to print intermediate results. Unlike function **peek**, however, one must use the **-glue** option to prevent removal of dead calls. We do recommend that this option only be used during debugging.

Another common source of trouble in SISAL programs is the mixed language interface itself. Argument mismatches are common, especially for calls with many parameters. Further, OSC will not verify that interface functions called from SISAL are reentrant and side-effect free; this is the programmer's responsibility. Also, one-dimensional SISAL arrays are passed by **reference**; and hence should be considered **read-only** in the interface routines. Arrays of greater dimensions, however, are passed by **value** and can be altered within the routines without disturbing SISAL execution.

Above discussions aside, the best way to track errors in SISAL programs is to use OSC's symbolic debugger (SDBX) which is discussed in Section 19.

18 Miscellaneous Features of OSC

In this section we briefly discuss two miscellaneous features of OSC. The first concerns intrinsic functions, and the second concerns the resolution of undefined externals.

18.1 Intrinsic Functions

OSC recognizes several global functions as special intrinsics and in some cases generates inline code. The following intrinsics manipulate integer data:

```
global and( A,B:integer returns integer )
global  or( A,B:integer returns integer )
global xor( A,B:integer returns integer )
global not(   B:integer returns integer )

global shiftr( A,B:integer returns integer )
global shiftl( A,B:integer returns integer )
```

Function `and` yields a bitwise and of arguments `A` and `B`, function `or` yields a bitwise or of arguments `A` and `B`, function `xor` yields a bitwise exclusive or of arguments `A` and `B`, and function `not` yields the one's complement of argument `B`. Function's `shiftr` and `shiftl` define bit position shifts. The former yields argument `A` shifted argument `B` bit positions to the right (possibly arithmetic). The later yields argument `A` shifted argument `B` bit positions to the left (with zero fill).

In addition to the above functions, OSC will recognize the following routines taken from the C math library (see math.h):

```
global sin( A:KIND returns KIND )
global cos( A:KIND returns KIND )
global tan( A:KIND returns KIND )
global asin( A:KIND returns KIND )
global acos( A:KIND returns KIND )
global atan( A:KIND returns KIND )

global sqrt( A:KIND returns KIND )

global log( A:KIND returns KIND )
global log10( A:KIND returns KIND )

global etothe( A:KIND returns KIND )  % exp in math.h
```

Type `KIND` can be one of `integer`, `real`, or `double_real`.

18.2 Resolving Underfined References

Because OSC supports mixed language programming, it also allows the specification of library search directives to help the local compilers resolve undefined references. In the following example, `libx.a` is searched during the load phase of compilation.

```
osc -forFORTRAN simple.f routines.sis -lx
```

By default, OSC will search the local C support libraries, including the C math library. For more information on this option see your local compiler users' guide or man-page.

19 The Symbolic Debugger: SDBX

In this section, we exemplify the functionality of OSC's symbolic debugger: SDBX. To use the debugger, simply compile the target module or program using the **-sdbx** option:

```
osc -sdbx -module -db dbase file1.sis file2.sis
```

Note that a module or program compiled for debugging will link and run with a program or module not compiled for debugging.

To illustrate SDBX and its commands, we transcribe the execution of the following program:

```
define main

function f( x:real returns real )
  x*x+1.0
end function

function main( start,finish:real; gran:integer returns real )
   let
      width := (finish-start)/real(gran);
   in
      for i in 1, gran
         x := start+((real(i-1))*width);
         y := f(x);
         v := y*width;
      returns value of sum v
      end for
   end let
end function
```

Function `main` computes the area under `x*x+1.0` between `start` and `finish` using granularity `gran`. We show the transcription below. The user provided commands follow the SDBX prompts, and the FIBRE input values for the run were 1.0, 2.0, and 3 respectively. Note that the break point commands are **set-and-go** in nature.

```
unix-> s.out input
SGI SISAL 1.2 V12.0
 [entering sdbx]
 [commands: bounds break cont functs help list
    names print quit run step where]
sdbx-> help
 bounds    NAME        [print the bounds of array NAME]
 break                 [continue and break just inside the next function]
 break     NAME        [continue and break just inside function NAME]
 break     return      [continue and break just before next function return]
 break     end         [continue and break at next scope end]
 break     LINE        [continue and break on completion of line LINE]
 cont                  [continue execution]
 functs                [list the name of each function in the module]
 help                  [this command]
 list                  [list the previously executed line]
 list      LINE        [list line LINE ]
 list      LINE1 LINE2 [list lines LINE1 through LINE2 inclusive]
 names                 [list all currently defined NAMES]
 print     NAME        [print the value of NAME]
 print     NAME  FILE  [append the value of NAME to FILE]
 quit                  [abort execution]
 run                   [identical to cont]
 step                  [execute the next line and return to sdbx]
 where                 [list all the active functions]
*special names:        [$NUM, scope or function result]
                       [#1, array of previous dereference]
                       [%1, denominator of previous division]
sdbx-> where
 [in program or module startup routine]
sdbx-> functs
 f          main
sdbx-> break
*   7: function main( start,finish:real; gran:integer returns real )
sdbx-> names
 start      gran       finish
sdbx-> print gran
 gran = 3
sdbx-> step
*   9:       width := (finish-start)/real(gran);
sdbx-> print width
 width = 3.333333e-01
sdbx-> step
*  12:          x := start+((real(i-1))*width);
sdbx-> print i
 i = 1
sdbx-> break f
*   3: function f( x:real returns real )
```

```
sdbx-> where
 [f,area.sis]
 [main,area.sis]
sdbx-> print x
 x = 1.000000e+00
sdbx-> break return
 [end function]
sdbx-> names
 x          $1
sdbx-> print $1
 $1 = 2.000000e+00
sdbx-> step
*  13:          y := f(x);
sdbx-> print y
 y = 2.000000e+00
sdbx-> list 11 16
   11:       for i in 1, gran
   12:          x := start+((real(i-1))*width);
*  13:          y := f(x);
   14:          v := y*width;
   15:       returns value of sum v
   16:       end for
sdbx-> break 14
*  14:          v := y*width;
sdbx-> print v
 v = 6.666667e-01
sdbx-> step
*  12:          x := start+((real(i-1))*width);
sdbx-> print i
 i = 2
sdbx-> names
 start      gran       finish     %1         width      i          x
sdbx-> break end
 [end scope]
sdbx-> list
*  14:          v := y*width;
sdbx-> print v logfile
sdbx-> break end
 [end scope]
sdbx-> break return
 [end function]
sdbx-> where
 [main,area.sis]
sdbx-> names
 start      gran       finish     %1         width      $1
sdbx-> print $1
 $1 = 2.851852e+00
sdbx-> cont
```

```
 [sdbx processing complete]
sdbx-> where
 [in program or module startup routine]
sdbx-> run
2.851852e+00
unix->
```

Note that "`print v logfile`" appended the value of `v` to file `logfile`.

We use the following program to illustrate the remaining commands: **bounds**, **print** array-name, and **quit**.

```
define main

function main( returns array[integer] )
  let
    a := array[1: 1, 2, 3, 4 ];
  in
    a
  end let
  end function
```

The SDBX transcription follows:

```
unix-> s.out
SGI SISAL 1.2 V12.0
 [entering sdbx]
 [commands: bounds break cont functs help list
    names print quit run step where]
sdbx-> step
*   3: function main( returns array[integer] )
sdbx-> list 1 9
    1: define main
    2:
*   3: function main( returns array[integer] )
    4: let
    5:   a := array[1: 1, 2, 3, 4 ];
    6: in
    7:   a
    8: end let
    9: end function
sdbx-> break 5
*   5:   a := array[1: 1, 2, 3, 4 ];
sdbx-> print a
 a =
[ 1,4: # DRC=1 PRC=1
  1
  2
```

```
    3
    4
  ]
  sdbx-> bounds a
   [lower=1,upper=4,size=4]
  sdbx-> quit
  killed
  unix->
```

SDBX will also trap to the monitor if control-C is entered. Other events that will return control to the monitor include arithmetic faults (division by zero, etc.) and subscript violations.

20 Getting and Installing Osc

The OSC software is available via anonymous `ftp`, and resides on sisal.lln.gov (128.115.19.65) under the name

```
~ftp/pub/sisal/osc.v12.0.tar.Z
```

A transcription of an installation processes follows. The target machine was a Cray X-MP/48:

```
unix-> pwd
/wrk/w1/cann/Sisal
unix-> ftp 128.115.19.65
Connected to 128.115.19.65
220 lll-crg.llnl.gov FTP server (Version 4.163 Fri Feb 23 1990) ready.
Name (128.115.1.1:cann): anonymous
331 Password required for anonymous.
Password:
230 User cann logged in.
ftp> binary
200 Type set to I.
ftp> cd ~ftp/pub/sisal
250 CWD command successful.
ftp> get osc.v12.0.tar.Z
200 PORT command successful.
150 Opening data connection for osc.v12.0.tar.Z (binary mode).
226 Transfer complete.
ftp> quit
221 Goodbye.
unix-> uncompress osc.v12.0.tar.Z
unix-> ls
osc.v12.0.tar
unix-> tar -xof osc.v12.0.tar
unix-> ls
Backend/        Man/            Runtime/        osc.v12.0.tar
```

```
Examples/       Manual/         Tools/          sinstall*
Frontend/       README          bin/
unix-> sh sinstall

* This script will ask some questions about your system and build a
* Makefile for osc (Optimizing SISAL Compiler) installation.
* If you already have a file called "Makefile" it will be overwritten!
* For some questions, a default response is given in [].
* Pressing RETURN in response to such a question will enable the default.
* Answer yes/no questions with y or n.

Is this system:
1.  Sun running SunOS
2.  Some other sequential machine running UNIX
3.  Sequent Balance running DYNIX
4.  Alliant FX series running Concentrix
5.  Encore Multimax running Umax
6.  Sequent Symmetry running DYNIX
7.  Cray X-MP or Y-MP running UNICOS
8.  Cray 2 running UNICOS
9.  SGI running IRIX
Enter a number: [1]
7

Enter the number of available processors
4

Optimize the installed code? [y]
y

Compile for run time dbx use via "-g"? [n]
n

Enter path to directory for executables: [/usr/local/bin]
/wrk/w1/cann/Sisal/bin

Enter path to man pages: [/usr/man/manl]
/wrk/w1/cann/Sisal/Man/manl

* Makefile construction in progress...

* Makefile has been built. Enter "make all" to build and install osc.
unix-> make all >& LOGFILE
```

In summary, OSC installation proceeds as follows:

1. Using anonymous `ftp`, acquire a copy of the compiler.

2. Move the compressed tar file to the destination directory on the target machine and unpack it:

```
uncompress osc.v12.0.tar.Z
tar -xof osc.v12.0.tar
```

3. Run `sinstall` to configure the installation.

4. Type "`make all >& LOGFILE`" to run the installation.

The Examples directory contains several Sisal programs along with documentation describing their compilation and execution.

20.1 Questions, Bug Reports, and Concerns

Send questions, concerns, and general SISAL commentary to

`sisal-info@sisal.llnl.gov`

Send bug reports to

`sisal-bugs@sisal.llnl.gov`

To be added to the SISAL mailing list, direct your request to

`sisal-info-request@sisal.llnl.gov`

Acknowledgements

I wish to thank Dr. Rod Oldehoeft for his support and guidance while I designed and implemented OSC. He also coauthored an earlier OSC manual.

Finally, I would like to thank my wife, Sue, for both her technical and moral support.

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] Mathew Burns. Sisal challenges Fortran. *Supercomputing Review*, 5(2):72–73, February 1992.

[3] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[4] D. C. Cann. Retire Fortran? A Debate Rekindled. In *Supercomputing 91*, November 1991.

[5] D. C. Cann, C. Lee, R. R. Oldehoeft, and S. K. Skedzielewski. SISAL multiprocessing support. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, 1987.

[6] D. C. Cann, R. Wolski, and J. T. Feo. SISAL: Toward resolving the parallel programming crisis. In *IPPS 1992 Parallel Systems Fair*, March 1992.

[7] J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–365, December 1990.

[8] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[9] P. H. Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[10] C. P. Hendrickson. Programming a real code in a functional language (part i). In *Visions of Supercomputing Conference*, September 1991.

[11] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[12] B. J. MacLennan. *Functional Programming Practice and Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[13] J. R. McGraw and T. S. Axelrod. Exploiting multiprocessors: Issues and options. Technical Report UCRL-91734, Lawrence Livermore National Laboratory, Livermore, CA, November 1984.

[14] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[15] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[16] R. R. Oldehoeft, D. C. Cann, J. T. Feo, and A. P. W. Bohm. The SISAL 2.0 reference manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, Livermore, CA, December 1991.

[17] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[18] T. R. Richert. Efficient task management for SISAL. Technical Report 89-111, Computer Science Department, Colorado State University, July 1989.

[19] S. Skedzielewski and J. Glauert. IF1—an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[20] S. K. Skedzielewski and R. Kim Yates. Fibre: An external format for SISAL and IF1 data objects, version 1.0. Manual M-154, Lawrence Livermore National Laboratory, Livermore, CA, January 1985.

[21] Lisa Stapleton. Livermore labs offers cray time—free. *Supercomputing Review*, 4(5):14–15, May 1991.

[22] M. L. Welcome, S. K. Skedzielewski, R. K. Yates, and J. E. Ranelletti. IF2: an applicative language intermediate form with explicit memory management. Manual M-195, University of California Lawrence Livermore National Laboratory, November 1986.

[23] R. Wolski, J. Feo, and D. C. Cann. A prototype functional language implemenation for hierarchical-memory architectures. In *25th Hawaii International Conference on System Sciences*, January 1992.

A OSC Man Page

NAME

osc - optimizing SISAL compiler

SYNOPSIS

osc [option]... [file]...

DESCRIPTION

OSC compiles SISAL programs and modules, and generates SISAL executables. Those command line arguments having the suffix '.sis' are taken to be SISAL source files; each is parsed, with its resulting object code (IF1) left in the file whose root name is that of the source file and suffix is '.if1'. Then all resulting and provided '.if1' files are linked to form a program (the default) or module for optimization and code generation. A program is callable from the Operating System (called 's.out' by default) or possibly an external language (see -forC and -forFORTRAN) with 'main' defining the default entry point (see -e). A module on the other hand, is only callable from a program or another module. The module system facilitates the construction of large SISAL applications (see -module and -db).

Compilation may be stopped after any compilation phase and resumed at a later time by simply providing the intermediate file(s) on the compilation command line. Unless otherwise specified, the results of compilation and any provided '.o' (compiled SISAL modules, etc.) and '.a' files and specified libraries (-l option) are loaded (in the order given) to produce an executable, which must include a single program.

By default, all optimizations are done: scalar, build-in-place, and update-in-place. The scalar optimizations include record and array fission, strength reduction, common subexpression removal, loop invariant removal, loop fusion, loop unrolling, loop unswitching, loop distribution, global common subexpression removal, constant folding, constant aggregate identification, and dead code removal. On concurrent machines, program concurrentization is the default. On vector machines, program vectorization is the default. On concurrent-vector machines, program concurrentization and vectorization are the defaults. On sequential machines, program parallelization is automatically disabled; however, a parallelized program will compile and execute on any machine.

Options:

-v	Verbose mode: echo the command lines invoking the various compiler phases.
-d	Debug mode: echo, but do not execute, the command lines invoking the various compiler phases.
-nocpp	Do not run the C preprocessor on the named SISAL files. Default: run the C preprocessor.
-CPP	Run each named SISAL file through the C preprocessor and stop, leaving each result in 'root.i', where 'root' is the root of the file.
-Dname[=val]	Define macro name to the C preprocessor as if it were defined by a '#define' directive in SISAL source (see cc(1)).
-Uname	Undefine macro name to the C preprocessor as if it were undefined by a '#undef' directive in SISAL source (see cc(1)).
-Ipath	Establish path as one of the first directories searched by the C preprocessor when expanding '#include' directives in SISAL source (see cc(1)).
-IF1	Compile the named SISAL files to IF1 and stop. No optimizations are done.
-listing	For each SISAL source file 'root.sis' and 'root.i' produce a listing and cross reference table and place it in 'root.lst'. Default: Do not produce listings and cross reference tables.
-MONO	Compile the named SISAL files to IF1, link all

the resulting and provided '.if1' files to form a monolithic program, and stop. The resulting monolith is placed in 'root.mono', where 'root' is the root of the first file on the command line. No optimizations are done.

-hybrid Compile the named SISAL files and generate both C and FORTRAN. The default is to generate C only. In some situations, this can increased performance, especially on the Crays. OSC places the resulting FORTRAN in 'rootF.f', where 'root' is the root of the first file on the command line. OSC will automatically compile the code using the local FORTRAN compiler (see -FF); however, it is the user's responsibility to provide the appropriate and desired compilation options, including optimization switches, etc. (see -ff). Note, OSC uses 'root' when building unique FORTRAN subroutine names, and OSC does not destroy 'rootF.f' files.

-C Compile the named SISAL files, generate C, and Stop. The resulting C is placed in 'root.c', where 'root' is the root of the first file on the command line. This option will preserve FORTRAN code generated due of the -hybrid option (in 'rootF.f').

-S Compile the named SISAL files, generate assembly language, and Stop. The resulting assembly language is placed in 'file.s', where 'file' is the root of the first file on the command line.

-noload Compile the named SISAL files, generate object code, and Stop. The resulting object code is placed in 'root.o', where 'root' is the root of the first file on the command line.

-o absolute Name the final executable program absolute. Default: s.out.

-e funct Take function funct as a program or module entry point (the default entry point is 'main'). Entry points can also be specified within SISAL text using the ''%$entry=funct_list'' pragma, where ''funct_list'' is a comma separated list of function names; blanks and tabs are allowed in the list. Unlike modules, only one program entry point is allowed. See IMPORTANT NOTES at the bottom of this man page.

-c funct Take function funct as a C external language function. C external language functions can also be specified within SISAL text using the ''%$c=funct_list'', where ''funct_list'' is a comma separated list of function names; blanks and tabs are allowed in the list. Default: take the function to be a module entry point reference. See IMPORTANT NOTES at the bottom of this man page.

-f funct Take function funct as a FORTRAN external language function. FORTRAN external language functions can also be specified within SISAL text using the ''%$fortran=funct_list'', where ''funct_list'' is a comma separated list of function names; blanks and tabs are allowed in the list. Default: take the function to be a module entry point reference. See IMPORTANT NOTES at the bottom of this man page.

-module Compile the provided files into a single module callable from other sisal modules and a program. Multiple module entry points are allowed (see -e). Default: compile the provided files into a single program callable from an external language program (see -forFORTRAN and -forC) or the operating system (the default).

-db database During module compilation, use database as the inter-module data repository, which OSC's manages and uses to improve inter-module efficiency. If used, the same database file must be supplied for each invocation of OSC used to build the final executable (including compilation of the program) Failure to do this will result in a load-error. The file should only be manipulated by OSC, although it is human readable. WARNING: OSC does not guarantee atomic access to to database.

-forC Compile the provided files into a program callable from C only. The default entry point is 'main' (see -e). In this mode, more than one entry point is allowed. Files ending with '.c' are passed to the local C compiler along with the other files required to finalize compilation: See IMPORTANT NOTES at the bottom of this man page. The resulting executable is by default 'c.out'.

-forFORTRAN Compile the provided files into a program callable from FORTRAN only. The default entry point is 'main' (see -e). In this mode, more than one entry point is allowed. Files ending with '.f' are passed to the local FORTRAN compiler along with the other files required to finalize compilation: See IMPORTANT NOTES at the bottom of this man page. The resulting executable is by default 'f.out'.

-noopt Disable all optimizations: shorthand for -noscalar, -noinline, -nomem, -noup, -novector, and -noconcur.

-noinline Skip function inlining. Default: all functions, except those part of recursive cycles and those that would cause an explosion of code, are inlined.

-inter Interactively select functions for inlining; ignored if -noinline is given.

-inlineall Inline all functions except those part of recursive cycles. This option conflicts with -inter and -noinline, but not -call. Default: Inline all functions, except those part of recursive cycles and those that would cause an explosion of code.

-call funct Do not inline function funct. Ignored if -inter is given. Default: Try and inline function funct.

-time funct Generate code to time the execution of function funct, reporting the data to stderr. This option implies "-call funct".

-noscalar Disable all scalar optimizations. This option implies -nofuse. Default: do all scalar optimizations.

-noifuse Disable independent loop and conditional fusion. Default: do independent loop and conditional fusion.

-nodfuse Disable dependent loop and conditional fusion. Default: do dependent loop and conditional fusion.

-nofuse This option implies -noifuse and -nodfuse. Default: do independent and dependent loop fusion (see -noscalar).

-noOinvar Disable the invariant removal of inner loops from the outer loops of each function. This is useful when benchmarking a computational kernel.

-uNUM Selectively unroll for loops having no more than NUM iterations. An unrolling value of zero disables loop unrolling as does -noscalar. Default: -u2.

-nounroll Disable loop unrolling. This option is equivalent to -u0 and is implied by -noscalar.

-nomem Skip all subphases of build-in-place analysis, including attempts to preallocate storage for arrays returned by product-form for loops. This option implies -nopreb. Default: do all subphases of build-in-place analysis.

-noup Disable all phases of update-in-place analysis. This option implies -nopreb.

-seq Compile for sequential execution. This option is equivalent to providing both -noconcur and -novector.

-pNUM If compiling for concurrent execution, partition the program to use no more than NUM processors. Default: Partition the program to use all the available processors : a number defined during OSC installation.

-nNUM If compiling for concurrent execution, only consider product-form for loops nested no deeper than NUM for concurrent execution. Default: consider all eligible product-form for loops regardless of nesting level. A NUM of 1 results in consideration of only the outermost loops. A NUM of 0 is equivalent to -noconcur.

-iNUM If compiling for concurrent execution, assume product-form for loops iterate an average of NUM times and arrays comprise on the average NUM elements. This number helps derive the execution cost estimates used during program concurrentization. Default: 100.

-hNUM If compiling for concurrent execution, only slice eligible product-form for loops with estimated execution costs greater than or equal to NUM. Default: 7000.

-maxconcur Shorthand for -h1, which results in compilation for maximal concurrency; that is, all for loops are assumed to have execution costs that warrant concurrent execution. The -n option can still be used to throttle concurrentization.

-noconcur Shorthand for ''-n0'', which disables concurrentization. Default: For concurrent machines, the default is to concurrentize loops.

-novector Disable loop vectorization. Default: For vector machines, the default is to vectorize loops.

-cray Cross compile the named program for execution on CRAY computers. This option implies -C.

-explode Aggressively apply loop distribution regardless of the cost in compilation time. Default: when compiling for vector execution OSC will do some loop distribution to uncover additional vectorization.

-explodeI Aggressively apply loop distribution regardless of the cost in compilation time, except only consider innermost loops (see -explode).

-icse Push identical operations sequences (possibly having nonidentical operands) between subgraphs down and out of conditionals. This option should be used with care as it will often increase program execution time on the Crays. It should improve execution time on machines that execute both branches of a conditional before discarding the unneeded results.

-noassoc Disable concurrentization and vectorization of loops comprising floating point reduction operations, and other associative transformations. Note, this option does not guarantee that the local C compiler will not do associative transformations (see -cc). Default: Attempt to concurrentize and vectorize loops defining floating point reduction operations, and do other associative transformations.

-pbNUM Set the array prebuilding dimension count to NUM. A value of zero disables prebuilding and a value larger than five is treated as five. Default: -pb3.

-nopreb Disable array prebuilding. This option is equivalent to -pb0.

-bind Declares that the descriptor data (lower and upper bounds, etc.) and array addresses passed each SISAL function call from C or FORTRAN will never changes between invocations. However, array components may change between calls. Use of this option can dramatically increase the performance of the interface during repetitive invocations. Default: Assume the descriptor data and array addresses may change between invocations.

-glue Disable the removal, hoisting, and combining of non-inlined function calls.

-bounds Generate code to check for and report array subscript violations, and other problems such as division by zero. This option implies -novector, -noinline, -glue, -noscalar, and -nopreb. Warning, this option may degrade the run time performance of the program or module being compiled. Default: do not check for subscript violations and other problems.

-sdbx Generate code to interface with OSC's symbolic debugger. Note that -bounds and -sdbx conflict, as one function of the symbolic debugger is to report subscript violations and divisions by zero. This option implies -novector, -noconcur, -noinline, -glue, -noscalar, -noup, -nomem, -nopreb, and -noimp. Warning, this option will degrade the run time performance of the program or module being compiled.

-copyinfo Generate code to gather aggregate copy information at run time and write the data to s.info (see the -r option in s.out.l). This option can result in increased program execution time. Default: do not gather copy information.

-flopinfo Generate code to gather floating point operation counts at run time and write the data to s.info (see the -r option in s.out.l). This option can result in increased program execution time. The operations counted are +, -, *, /, and negation, and reduction operations, logicals, and floating point intrinsics. Filtered reductions are not counted. Default: do not gather floating point operation count information.

-fflopinfo funct
Generate code to gather floating point operation counts at run time for function funct and write the data to stderr. This option implies -flopinfo. Default: do not gather floating point operation count information for function funct.

-noimp Compile with the C compiler's optimizers disabled (see cc(1)). Default: Compile with the C compiler's optimizers enabled.

-CC=COMPILER Use COMPILER to compile C source.

-cc=OPTION Give option OPTION to the local C compiler. For example, to profile SISAL execution, ''-cc=-pg'' works for most Unix C Compilers.

-FF=COMPILER Use COMPILER to compile FORTRAN source (see -forFORTRAN).

-ff=OPTION Give option OPTION to the local FORTRAN compiler (see -forFORTRAN).

-real Treat all SISAL double_real data as real data. Default: honor program declarations. SISAL real maps to C float on the target machine. On the Cray, SISAL double_real maps to C float.

-double_real Treat all SISAL real data as double_real data. Default: honor program declarations. Except for the Cray, SISAL double_real maps to C double. On the Cray, all double_real data is treated as real (see -real), and this option is ignored.

-info Print diagnostic information gathered during compilation to stderr. Default: run silently.

-cvinfo Print concurrentization and vectorization information gathered during compilation to stderr. This option provides a subset of the information generated by the -info option. Default: run silently.

-lx During the load phase of compilation, search library 'libx.a', where x is a string, to resolve undefined externals. Refer to ld(1) for the default search paths.

INTRINSICS

OSC recognizes intrinsic functions "and" (bitwise and), "or" (bitwise or), "xor" (bitwise exclusive or), "not" (one's complement), "shiftl" (left shift with zero fill), "shiftr" (right shift, possibly arithmetic), and the following math functions taken from the C math library (see math.h): sin, cos, tan, asin, acos, atan, sqrt, log, log10, etothe (exp in math.h).

IMPORTANT NOTES

If a SISAL program builds an error value at run time, the program will print an error message and abort; further, ''is error'' always yields ''FALSE''. Stream data types are processed as array data types. Program entry points are not reentrant with respect to the outside world. Further, no guarantee is given that Fortran or C calls from SISAL are reentrant. To facilitate program debugging, OSC supports a predefined function peek that accepts zero or more arguments of any type and returns integer 1. Peek prints its input values to standard error in FIBRE format. A call to this function will not be removed by dead code elimination.

B S.OUT Man Page

NAME

s.out - SISAL executable

SYNOPSIS

s.out [options] [infile] [outfile]

DESCRIPTION

S.out is the output file of the optimizing SISAL compiler (see osc(1)) and the link editor ld(1). It is executable if there were no errors and no unresolved externals during compilation. When execution begins, FIBRE input is read from infile (if provided) or standard input and is associated positionally with the arguments of the main SISAL function. At completion, results are written in positional order to outfile (if provided) or standard output in FIBRE form. The character '-' implies standard input or output if it appears as a file parameter.

Options:

-wNUM Set the number of worker processes to NUM. A value greater than one makes sense only on a multiprocessor SISAL implementation (Cray, Encore, Sequent, Alliant, etc.). Default: 1.

-lsNUM If compiled for concurrent execution (refer to osc(1)), slice each concurrentized for loop into

NUM pieces. Default: Slice each concurrentized loop into W pieces, where W is the number of workers.

-gss If compiled for concurrent execution, slice each concurrentized for loop using guided self scheduling. Here the thickness of a slice is dynamically computed as the number of iterations not yet scheduled divided by the number of workers. Note that the -ls and -gss options conflict.

-b If compiled for concurrent execution (refer to osc(1)), use the distributed run queue system to bind parallel work to worker processes in a consistent and regular manner. In the absence of gang scheduling, this option could degrade performance if the machine load is high. Default: Use the shared run queue system. WARNING: on the SGI this option binds processors to worker processes, and under heavy system loads, this can severely degrade job and system throughput.

-axNUM Set array expansion value to NUM. Each time the space for a dynamically growing array must be expanded, it will obtain NUM times the number of previous expansions more elements. Programs with dynamically growing arrays may benefit from a larger value. The expansion value is also used to prevent memory fragmentation in programs repeatedly building and recycling incrementally smaller or larger arrays. Default: 100.

-dsNUM Initialize the shared data management pool to NUM bytes. An execution that terminates because of storage deadlock may need more dynamic memory. Default: 2000000 bytes.

-dxNUM Set the exact fit storage allocation threshold to NUM bytes. This eliminates the existence of leftover free blocks whose sizes are smaller than NUM bytes. Default: 100.

-r Append resource utilization information to the file s.info (elapsed cpu time, elapsed wall clock time, memory utilization figures, etc.). A new file is created if s.info does not exist.

-z Do not print the program's output. Default: print the program's output.

IMPORTANT NOTES

One may provide an upper bound in a FIBRE array definition to reduce FIBRE processing time. Simply follow the lower bound by a comma and then the upper bound; for example, "[1,2: 10 20]".

C SPEEDUPS Man Page

NAME

speedups - SISAL parallel speedups data gatherer

SYNOPSIS

speedups strtnr endnr abs [options] [infile] [outfile]

DESCRIPTION

Speedups repeatedly executes the compiled SISAL program ''abs'' using different numbers of processors, beginning with ''startnr'' and incrementing through ''endnr.'' For each execution it automatically appends the ''-wNUM'' option (overriding any of your own). It also enables ''-r'' so that each execution produces file s.info. When speedups completes, file ''$abs:t.info.$strtnr-$endnr'' will contain the concatenation of all the performance data.

Disclaimer