Bull. Nov. Comp. Center, Comp. Science, 35 (2013), 101–119

Back-end translator for Sisal 3.1 compiler

K. A. Pyzhov, R. I. Idrisov

Abstract. Sisal is a single-assignment language without side effects. Sisal supports error values and a flow data type and allows recursion. It has a verbose syntax, which reminds that of Pascal in some cases. Sisal is positioned as a language for scientific computations, implicitly parallel and effective. It can significantly simplify the parallel program development if the same program is intended both for a cluster and for one executor, for big and small data effectively. Here we describe the key features of our back-end translator.

Key words: functional programming, compilers, program optimization.

1. Introduction

Sisal is a pure functional language initially developed at the Lawrence Livermore National Laboratory [1]. In this paper, we describe a back-end translator for the updated language version 3.1 developed at our institute (IIS SB RAS).

The main difference between imperative and functional paradigms is the state of memory which is absent in pure functional languages. There are no named "memory cells" (variables) and no state changes (side effects). Any sub-program or operator (in terms of the imperative paradigm) must return some value as a result. It makes the code implicitly parallel because there are no concurrent variable sets or data-races.

Lack of side effects makes the dependence analysis and parallelization for functional languages much easier than for imperative languages. There is no need to build a separate dependence graph for analyzing the applicability of certain optimizing transformations, parallelization and vectorization. All data dependencies are represented explicitly in an intermediate representation for data-flow languages.

Of course, any imperative program can be converted to the Static Single Assignment (SSA) form to exclude variable re-assignment (in this case imperative variables or memory-cells can become "values" and some parts of the analysis will be simplified) but the semantics of the whole program remains imperative [2]. Such conversion is used in real compilers. For instance, the SSA form ([3], [4]) significantly simplifies application of data-flow optimizations for scalar variables in the intermediate representations of imperative programs.

Sisal is a single-assignment language without side effects [5]. Sisal supports error values and a flow data type and allows recursion. It has a verbose syntax, which reminds that of Pascal in some cases. Sisal is positioned as a language for scientific computations, implicitly parallel and effective. It can significantly simplify the parallel program development if the program is intended both for a cluster and for one executor, for big and small data effectively. In this text we describe the key features of our back-end.

2. A brief history of Sisal language versions and implementations

The initial point of Sisal is 1983, when several organizations (Livermore National Lab, Colorado University, Manchester University and DEC corp.) developed the language standard [1].

The first compiler was under development from 1985 to 1989. It was called OSC (Optimizing Sisal Compiler) and supported Sisal 1.2 [6]. Later, that compiler became a base for many improvements. Particularly, POSC (Partitioning and Optimizing Sisal Compiler) was presented in 1990. Unlike OSC, this compiler was able to divide a program into parallel tasks using the dynamic profiling.

In 1991, a new language version Sisal 2.0 was published but not implemented though. The next version, Sisal 90, has not been implemented as well.

We started working on Sisal 3.0 at the A.P. Ershov Institute of Informatics Systems in 2001 [7]. Currently, the Sisal language translator is used mostly by its developers for scientific purposes: developing new optimization and analysis algorithms, checking and improving the language standard.

3. Back-end translator

3.1. High-level scheme

The high-level scheme of the Sisal 3.1 compiler is presented in Figure 1. Its front-end translates a source program to IR1 (IR stands for an intermediate representation). IR1 and IR2 are the front-end and back-end high level graph representations; IR3 is the three-address code representation.

When the IR1 graph is built, it is sent to the back-end. The back-end translator includes the following phases:

- IR1 $\rightarrow$ IR2 translation,
- IR2 optimization,
- IR2 $\rightarrow$ IR3 translation,
- IR3 optimization,

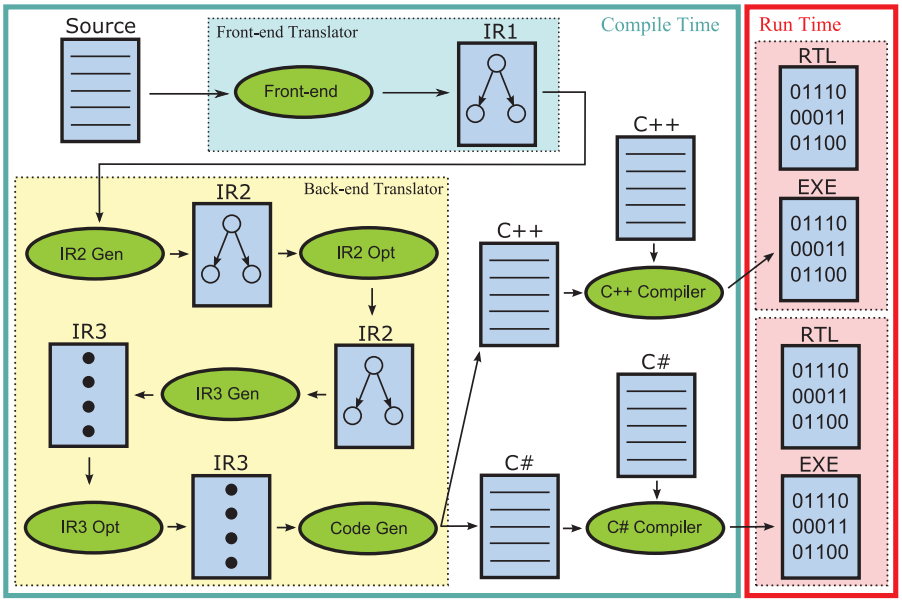

Figure 1. The Sisal compiler and run-time support

- output source generation.

3.2. Intermediate representation IR2

3.2.1. General description

IR2 is the Sisal program intermediate representation based on directed multi-graphs.

The IR2 graph for a Sisal function is a set $(G, VAR, \sigma_v, \preceq_e)$, where

- $G =< N, P, E >$ is a multi-graph that consists of a set of nodes N, a set of ports P and a set of edges E;
- VAR is a set of variables;
- σ_v is a mapping $E \to VAR$ which defines the correspondence between variables and graph edges;
- $\preceq_e$ is $N \times N$ ordering which defines the execution priority.

The set of nodes N includes the nodes of two types: simple and compound. The compound nodes may contain child nodes and represent the compound statements of the Sisal language (function, conditional expression, and loop).

Any node $N_i \in N$ in the graph G corresponds to two subsets of ports from P: "in" ports ($P_i^{in} \subset P$) and "out" ports ($P_i^{out} \subset P$). Any port belongs to only one node; we call this node "parent" for this particular port.

Any edge represents a value transfer and connects two ports. If one of these ports belongs to P^{in} and the other to P^{out}, we call parent nodes "directly connected". We call these nodes connected by output if both of the corresponding ports belong to P^{out}.

An edge is *inner* for a port if it connects a compound node with its internal nodes. It includes the edges connecting the node with itself or with its child nodes (Figure 2).

We call two nodes "connected" if they are directly connected or one of them is directly connected to a node which is connected with the other.

We call a set of nodes Q "directly nested" for the node n if $\forall q$, where $q \subset Q$, $\exists q'$ connected with q and connected by output with n.

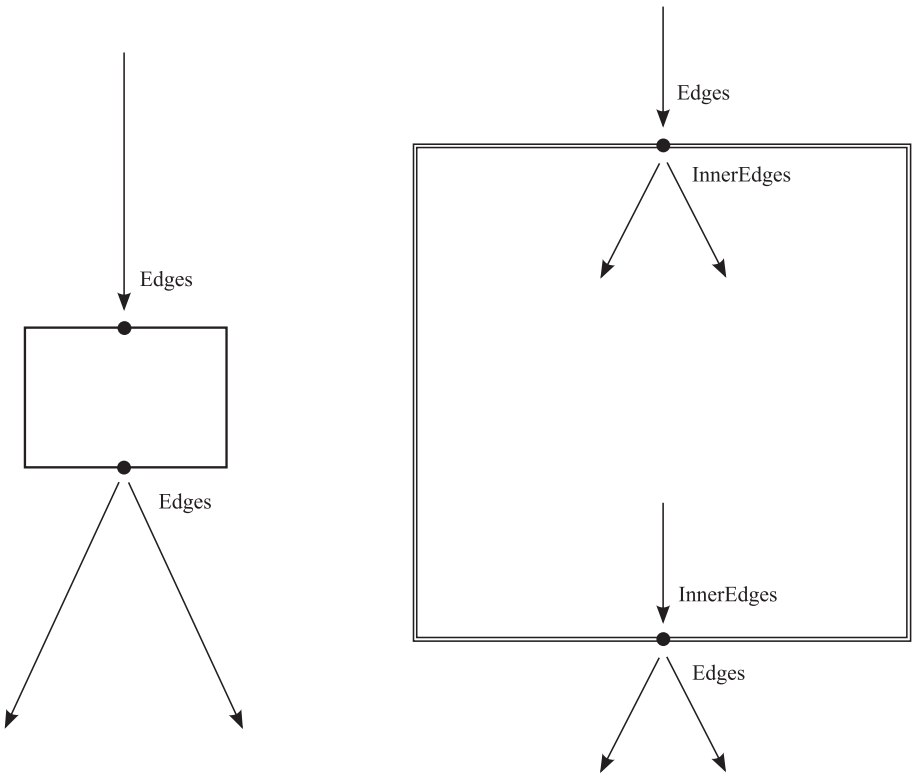

Figure 2. *Edges* and *InnerEdges*

The mapping σ_v maps each edge $E_i \in E$ to a variable $V_j \in VAR$.

The IR2 graph implicitly contains a hierarchy defined by the compound node semantics and the "directly nested" property. The inner nodes are usually visualized inside the parent nodes; the parent-child relation in this

case strictly corresponds to nesting in the source program. For example: "if case", "then branch" and "else branch" are the inner nodes for their "if" node. Such a visualization makes the IR2 representation more clear for a user because it corresponds to the code formatting style (when the number of spaces before a function indicates its nesting level). It is planned to be used for application debugging. Below we refer to the representation as hierarchical.

All edges in the IR2 graph are partially ordered by the $\preceq_e$ ordering which is based on the data flow of a Sisal program: if there is a path from a node N_1 to a node N_2, and N_1 and N_2 have the same nesting level (are directly nested for the same compound node), then $N_1 \prec_e N_2$.

If $N_1 \prec_e N_2$, then N_1 must be executed before N_2. If $N_1 =_e N_2$, then N_1 and N_2 can be executed in any order, and a parallel execution is possible. The ordering binding algorithm is shown in Figure 4.

The IR2 representation is built for the whole module and includes IR2 graphs for all functions of the module. These graphs are not connected.

Example 1. *Figure 4 shows the IR2 representation for the following fragment of the Sisal program:*

```
function sign(N: integer returns integer)
  if     N > 0 then  1
  elseif N < 0 then -1
  else               0
  end if
end function
```

3.2.2. Representation reducibility

The main function of the back-end internal representation is to provide a natural and usable structure for optimizations. In this part of the article, we consider connections and possible IR2 conversions to another well known computation model.

The block diagram transformations are widely used for imperative program optimizations [8]. IR2 cannot be translated to this scheme without a potential loss of its parallel properties because this diagram does not allow partial order for the instructions inside one linear block.

The most general representation for a parallel algorithm is Karp-Miller schemata [9], A scheme is defined as $S = (M, A, C)$, where A is a set of program operators, M is a memory cell set and C is an automat which drives the operator start and finish. Such a scheme has two elements which are not natural for a dataflow execution model: the automat for execution driving and the memory cells. In the dataflow model, execution starts when all operands of the function are ready (for a greedy model) or when the result

```
exec_priority_for_node(node N) {
    if (node N has subnodes) {
        N.priority = 1;
        exec_priority_for_subnodes(N);
    }
}

exec_priprity_for_subnodes(node N)
{
    for each subnode Q of node N {
        Q.priority = 1;
    }
    for each subnode Q of node N {
        exex_priority(Q);
    }
    for each subnode Q of node N {
        exec_priority_for_subnodes(Q);
    }
}

exec_priority(node N)
{
    int n = N.priority;
    for each "out" port P of node N {
        for each edge E from set Edges of port P {
            node Q = sink node of E;
            if (Q is not parent of N) {
                if (Q.priority <= n) {
                    Q.priority = n + 1;
                }
                exec_priority(Q)
            }
        }
    }
}
```

Figure 3. The $\preceq_e$ ordering algorithm

is required and the operands are ready (for a lazy model). So the transformation to Karp-Miller schemata will increase complexity of the analysis and optimization because it is not natural for the dataflow. The same is for the A-scheme [10] and the counter-operator subclass of the Karp-Miller schemata.

The dataflow parallel execution models are more natural for Sisal, but usually such models contain a lot of elements which are not used by IR2 and can describe only one set of the directly nested elements (do not support hierarchy). We mention here the dataflow Karp-Miller schemata [11], Adams [12] and Rodriguez [13] models.

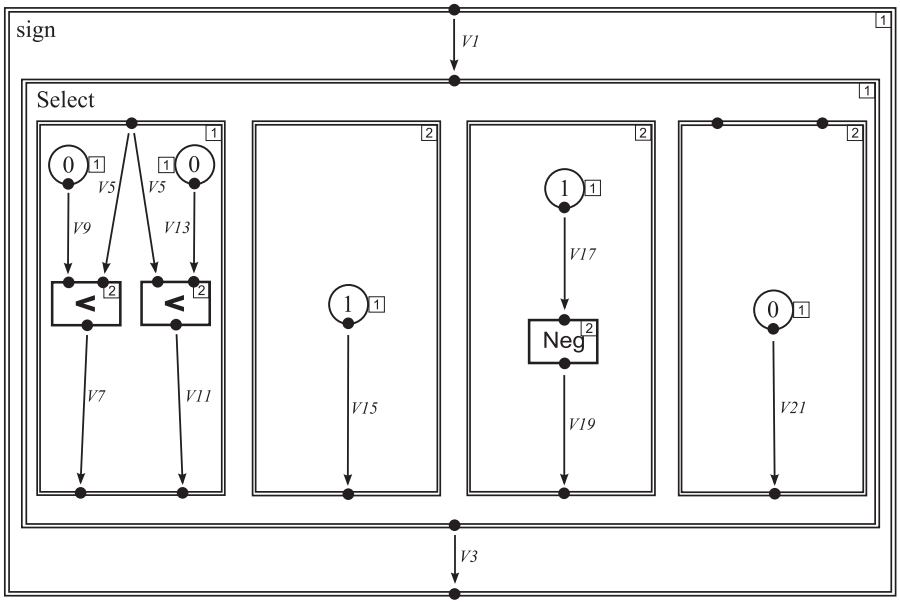

Figure 4. The IR2 representation

The most suitable hierarchical representation is the structured Petri nets [14] which can be used in investigation of some graphs without implicitly structured nodes like a loop. In IR2, a loop contains a loop body and reduction subnodes; these nodes are not connected because the actual dataflow structure of a loop depends on the input data.

3.2.3. Types and variables

For IR2 and IR3, we define the objects *variable* and *type*. A variable describes a Sisal object within the representations IR2 and IR3. In the IR2, variables are associated with the graph edges; in the IR3, variables are the operands of IR3 operations. Each variable has the following attributes: a unique identifier, a unique name, a type and an additional boolean variable which defines the "IsError" property.

The error property provides a more natural error handling for the parallel execution when compared to a popular try-catch model, when the execution must be stopped at some point and the exception code must be executed to handle the error. Such a model is not really functional because it contains some state and considers the operations to be ordered or uses a non-deterministic model otherwise [2].

All variables are divided into scalar variables, array variables and record variables. Each of these groups has additional properties (Figure 5).

Scalar variables have the size property. The array variables have three

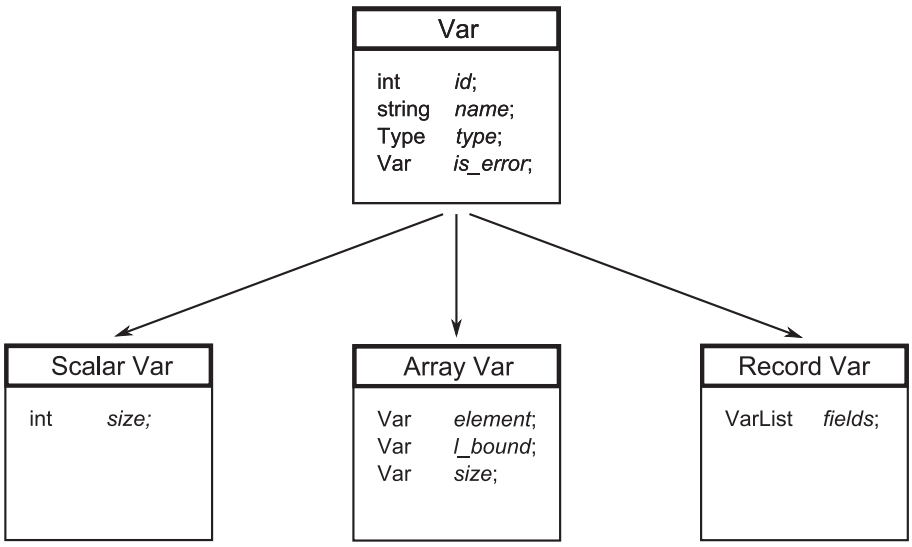

Figure 5. Variables

additional associated variables: an element of the array, a lower bound of the array and the size of the array. The record variables have an associated list of variables describing the fields of the record.

The types in IR2 and IR3 represent the types of the Sisal language within IR2 and IR3. A type contains additional low-level information about objects (such as machine representation of the type).

3.2.4. Optimization metadata

During the optimization process, the algorithms can create additional data connected with a node, an edge or a port (Figure 6). The data created by one algorithm can be reused by another. Below we call the function parameters aggregate metadata as "function call conditions" when these data are propagated. The metadata connected with a particular port usually mean the range of possible values known at the static analysis phase. In the current back-end implementation, we have different data description algorithms: based on single constant values, multiple constant values, a single range or a set of ranges. Particularly, these data are used in the dead code elimination to find some branches or nodes of IR2 which do not affect the resulting value.

In the example in Figure 6, analysis has discovered that the function always has the value "1" and therefore can be replaced by a constant.

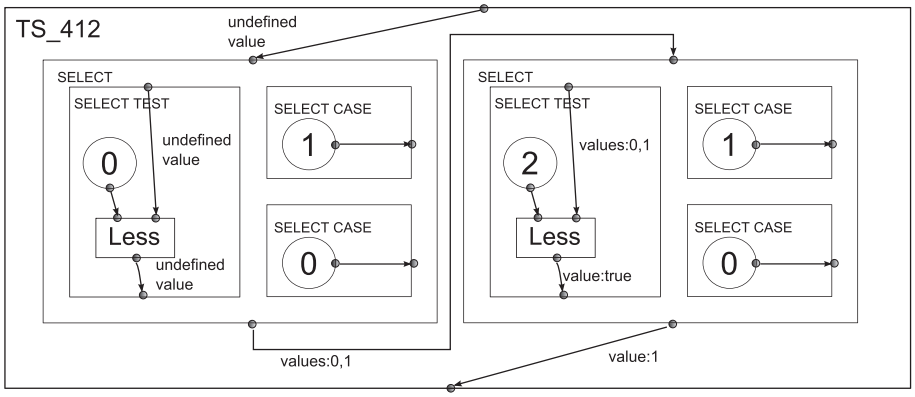

Figure 6. An example of function metadata

3.2.5. Translation of IR1 to IR2

IR2 contains IR1; therefore building the IR2 nodes and edges is trivial. To translate from IR1 to IR2, we set a partial order and assign variables to the edges.

3.2.6. Optimization effectiveness

To check the effect of complex optimizations, we estimate the optimized function run time in terms of the unlimited parallelism [15]. This concept assumes that we have the unlimited number of execution units sharing the same memory and performing any number of operations in parallel. Any directly nested subnodes of a compound node with their connections to each other can be considered as a data-flow graph, but if we assume the computation complexity of such nodes as equal, our analysis will lose any sense, because any function can be represented as a node. Its computation complexity will be equal to that of its inner nodes.

A strict time schedule for the data dependence graph G of an imperative program is a function defined for all graph nodes which is always ascending when traversing the graph edges: if there is an edge from a node u to a node v, then $f(u) < f(v)$. The vector h_i defines the computational complexity of particular operations and w_{ij} is the data transfer delay between the nodes i and j; h_i and w_{ij} depend on the properties of computation system. The difference between $f(u)$ and $f(v)$ cannot be less than operation complexity of u plus the data transfer delay between u and v (the sum of the corresponding values of h_i and w_{ij}). The vector of initial conditions s_i sets the time schedule function value for input nodes. It means that if u is the input node, then $f(u)$ is equal to the corresponding value of s_u. To estimate the function complexity in terms of the unlimited parallelism, we need to minimize the maximum value of the function f.

In IR2, graph nodes are connected via ports and edges, and the time schedule function should be defined for ports. Compound nodes of IR2 can be analyzed separately without any increase of the resulting time schedule function, but it requires an additional proof. The proof is not very complex but long and formal; we decided to exclude it from this paper. This proof makes possible to use the time schedule function analysis to estimate complexity of the function represented by the IR2 graph in terms of unlimited parallelism.

To conclude this part of the paper: when a complex optimization or a set of optimizations are made, we check the minimized maximum of the schedule function and if this value is greater than it was before optimization, it should be considered as inefficient.

3.2.7. IR2 optimization

IR2 contains implicitly connected graphs for different functions of the source program. To optimize the whole program as an ordinary dataflow graph, we need to put all functions on the same hierarchical level. It is not always possible, that is why we optimize function graphs separately and use function cloning when the call conditions are affecting the optimization results. The main structure for such optimizations is "Static call execution graph".

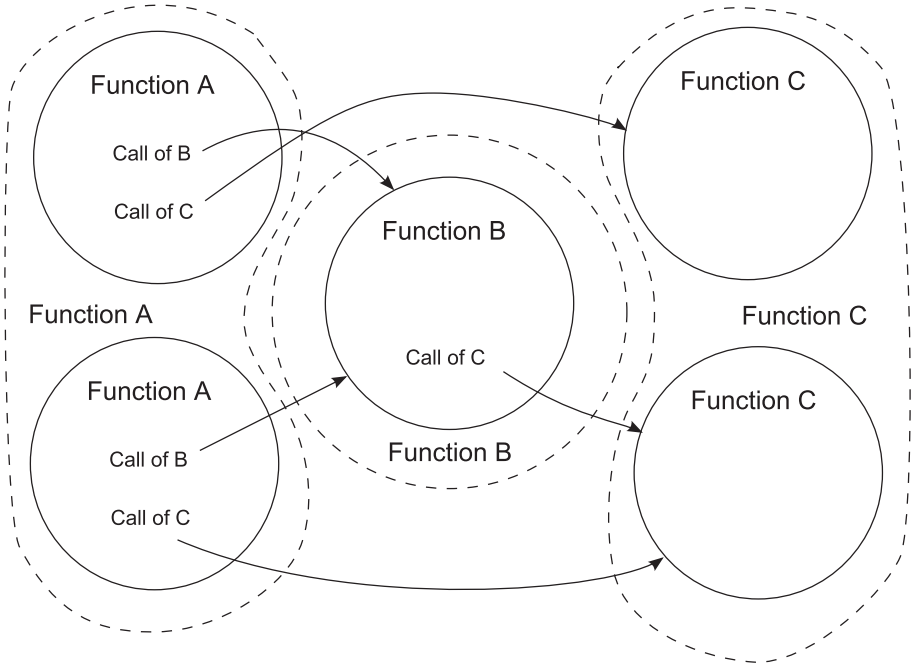

Figure 7. Static call execution graph

The static call execution graph contains functions and their copies as a hierarchy (Figure 7). It is not similar to memoization [16] because in memoization the results of execution of some function are stored for reuse. In our method, we collect similar initial conditions together and differentiate the executed entities when the optimizations of the called function are affected (estimated by the schedule function in terms of unlimited parallelism).

We use the term "interprocedural analysis" in the sense of the function dataflow analysis. In the functional paradigm, we have no memory state, all functions are pure and there are no imperative "procedures".

In the current version, the interprocedural analysis is an iterative algorithm which has two different strategies: *copying* and *backtrack*. The first strategy creates function copies while the performance is improved (with a restriction to avoid endless or very deep recursion), the second joins any conditions for the function without creating additional copies.

The following set of optimizations is applied to any analyzed function of the "Static call execution graph":

- optimization of the variable allocation for aggregate variables;
- moving the invariant computations out of loops;
- dataflow analysis and constant propagation;
- dead code elimination.

Optimization of variable allocation. Optimization of variable allocation allows us to avoid the redundant copying of aggregate objects. This transformation is done in two passes:

1. Finding the nodes that are the only users of aggregate objects;
2. Attaching the same variable to the "in" and "out" edges of nodes found at the first pass.

Example 2. *Consider the following fragment:*

```
function foo(A:array[array[integer]]; N:integer returns array[array[integer]])
  for i in 1, N cross j in i, N
  repeat
     A := old A[i,j := i,j]
  returns value of A
  end for
end function
```

Invariant code motion. The IR2 graph for the function foo has two nodes AReplace inside the nested loops. After IR2 has been built, both of these nodes have unique variables attached to the "in" and "out" ports. It means that each iteration of the loop will create two copies of the array A. But after applying the optimizing transformation, "in" and "out" edges of

AReplace nodes will carry the same variables (since both AReplace nodes are the only uses of their entries).

The invariant code motion is the transformation of the IR2 graph that moves from the loop body all nodes that can be executed outside the loop.

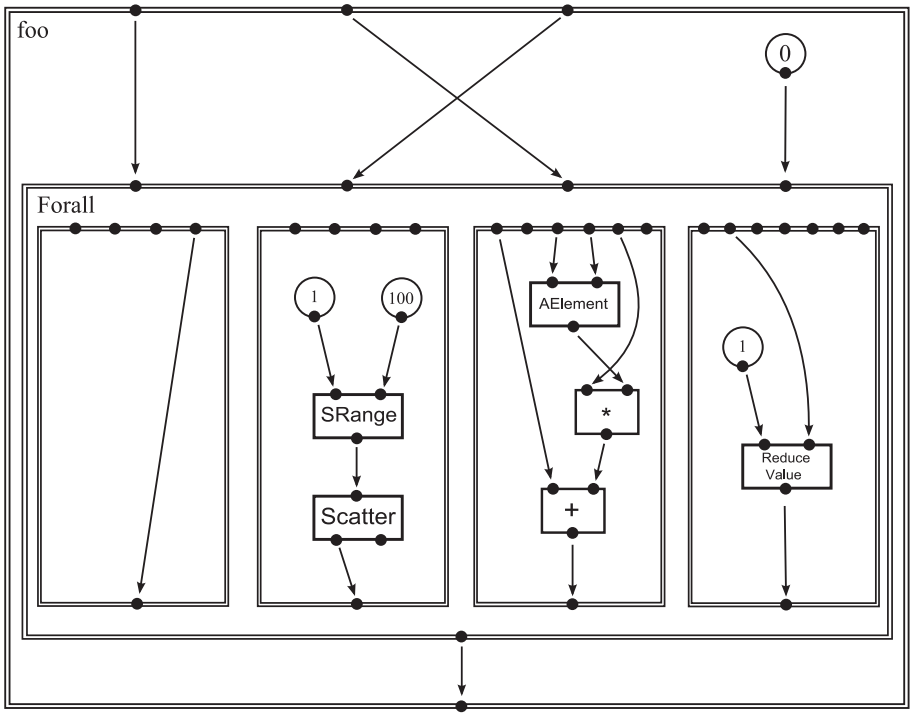

Figure 8. IR2 for the function foo

Example 3. *Consider the function:*

```
function foo(A:array[integer]; k:integer; N:integer  returns integer)
  let
    s := 0
  in
    for i in 1,100
      repeat
        s := old s + k * A[N]
      returns value of s
    end for
  end let
end function
```

Figures 8 and 9 show the IR2 representation before and after applying the invariant code motion.

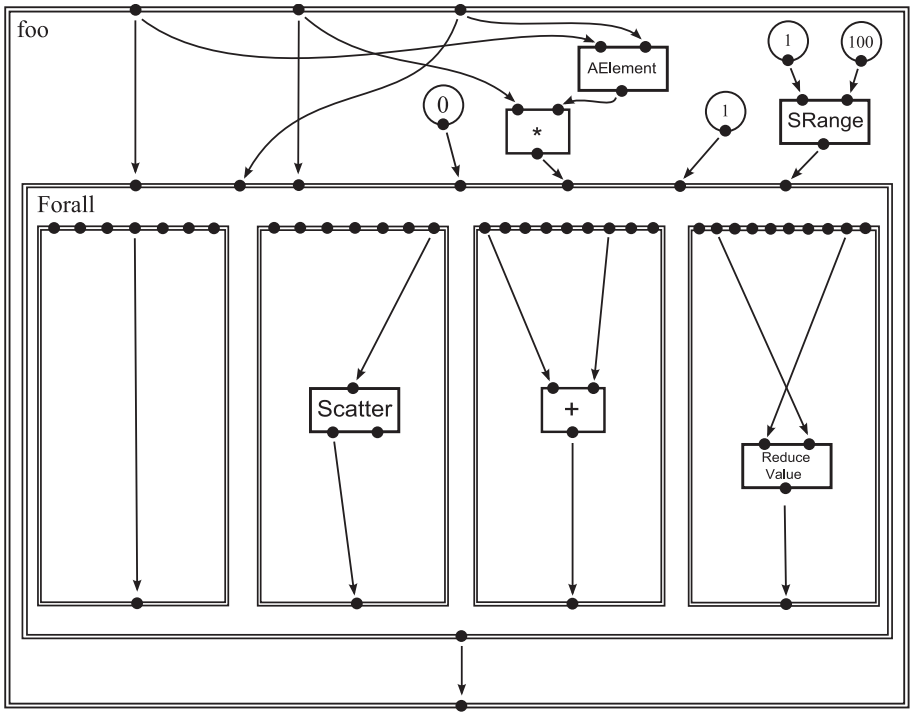

Figure 9. IR2 for the function foo after the invariant code motion

Dataflow analysis and constant propagation. This algorithm is repeated iteratively for the IR2 nodes down the hierarchy to gather all the information available at compile-time. As it was mentioned before, we have two different strategies for function cloning. In the dataflow analysis, we expand the "static call execution graph" and evaluate conditions for each function. When the analysis is stopped, all ports known to have a constant value and not connected to the constant are connected with the newly created constant node.

Dead code elimination. During this phase, all nodes which are not connected with the output ports are deleted. As it was already mentioned, our IR2 graph is not similar to the plain dataflow graph. Such dead nodes usually appeared after other optimizations.

3.3. Intermediate representation IR3

3.3.1. General description

IR3 is a classical three-address code representation with hierarchical blocks.

Example 4. *The content of IR3 for a function is listed below:*

```
 0    entry "function sign[integer]" (V_1(I32)  returns V_3(I32));
      {
 1        V_5(I32) = V_1(I32);
 2        V_5(I32) = V_1(I32);
 3        V_9(I32) = 0x0(I32);
 4        V_13(I32) = 0x0(I32);
 5        V_7(BOOL) = (V_9(I32) < V_5(I32));
 6        V_11(BOOL) = (V_5(I32) < V_13(I32));
 7        if (V_7(BOOL) == true(BOOL))
          {
10            V_15(I32) = 0x1(I32);
11            V_3(I32) = V_15(I32);
          }
          else
          {
12            if (V_11(BOOL) == true(BOOL))
              {
15                V_19(I32) = 0x1(I32);
16                V_17(I32) =  - V_19(I32);
17                V_3(I32) = V_17(I32);
              }
              else
              {
18                V_21(I32) = 0x0(I32);
19                V_3(I32) = V_21(I32);
              }
          }
20        return;
      }
```

3.3.2. Translation IR2 → IR3

In this phase, the IR2 graph is translated to the sequence of IR3 statements.

IR3 is built in the following steps:

- Generating a sequence of the IR3 statements for each IR2 node.
- Inserting the obtained sequence to the whole IR3 statement sequence.

Also, at this stage all variables considered to be "safe" (cannot get the error value at the execution time) are unwrapped with the IsError property because this significantly affects the performance. A program at this level becomes imperative and contains variable re-assigns; the variable without IsError property can be set to the variable with this property and must be explicitly converted.

When some target execution becomes parallel, the operations ThreadFork and ThreadJoin are created. The current implementation supports only .NET SMP with heavyweight threads and have no runtime thread scheduling.

3.3.3. Optimization of IR3

The IR3 optimization block applies the following optimizing transformations to IR3:

- Copy propagation;
- Dead code elimination.

After optimization of IR3 has been performed, IR3 lowering is executed. It replaces the high-level intrinsic functions by a sequence of statements or a sequence of lower level intrinsic function calls that are inlined at the code generation phase.

Example 5. *The result of the IR3 optimization for the function sign is shown below:*

```
 0    entry "function sign[integer]" (V_1(I32)  returns V_3(I32));
      {
 5        V_7(BOOL) = (0x0(I32) < V_1(I32));
 6        V_11(BOOL) = (V_1(I32) < 0x0(I32));
 7        if (V_7(BOOL) == true(BOOL))
          {
11            V_3(I32) = 0x1(I32);
          }
          else
          {
12            if (V_11(BOOL) == true(BOOL))
              {
17                V_3(I32) =  - 0x1(I32);
              }
              else
              {
19                V_3(I32) = 0x0(I32);
              }
          }
20        return;
      }
```

3.4. Translating IR3 to C#

The current target platform for the Sisal 3.1 compiler is .NET. The translator generates the C# code. It allows the users to perform the experimental execution of Sisal programs and examine the effectiveness of optimizing transformations applied by the compiler.

3.5. Results

We have translated some of the Sisal 1.2 programs available in the Internet to Sisal 3.1 in order to check our compiler optimizations. The first example is the Fourier discrete transform program shown in Figure 10.

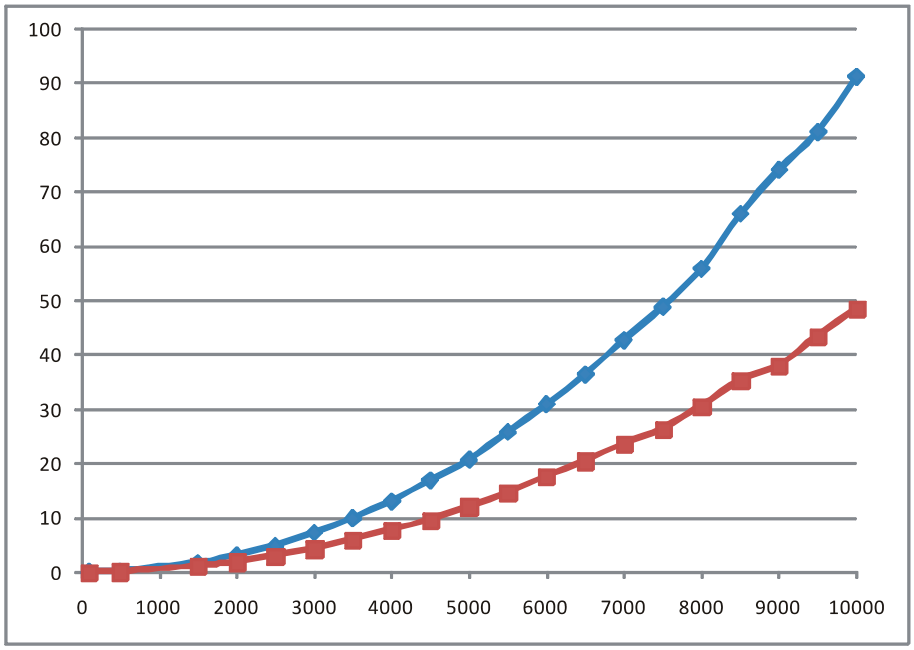

Figure 10. The result of DFT optimization: X is the array size, Y is the time in seconds; the lower curve (square markers) represents the optimized version

Speed improvement for the *DFT* program mostly caused by the data analysis has marked some of the variables as "safe" because the values cannot produce any error (no division by zero and no operation can take it out of the range). For other values we need to check for an error in every operation, because there are no native (supported by .NET) variables with the "error" property in C#.

The second example is a piece of a large aerodynamics code obtained from NASA Ames by J. Dennis created for the project "Mapping Array Computations for a Dataflow Multiprocessor". The function operates on three-dimensional arrays generated at runtime. In this example, we compare the results with interprocedural analysis and without it. Here three-dimensional arrays passed as the function argument and its value range can be estimated at compile time, that is why our fully optimized program shows good performance (Figure 11). In other words, a big part of the code was considered as useless and replaced with one constant. Of course it is a rare case and such a performance increase was achieved due to a non-optimal input program.

And the last figure shows a change in the memory usage, it was not big for the *DFT* task because the .NET platform allocates memory in big pieces and sometimes the optimization does not change the external amount

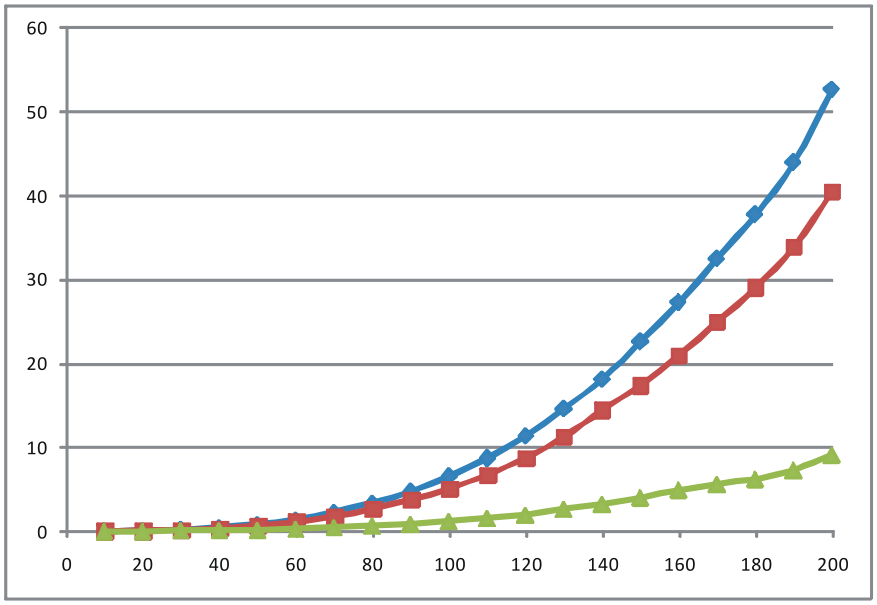

Figure 11. The smooth optimization result: X is the array size and Y is the time in seconds; the lower curve (triangle markers) represents the version optimized with interprocedural analysis; the middle curve (square markers) represents the version optimized without interprocedural analysis

of memory even if the number of variables has been changed. For this task we have a strange "step" in the center of the chart (Figure 12); we suppose that this "step" is caused by the features of the .NET allocation algorithms.

4. Conclusion and future work

The most part of the work done is theoretical. We have generalized the time schedule function method for estimation of the IR2 program execution. Our optimizations are proved not to increase the time schedule function. The algorithm of interprocedural data-flow analysis based on the "static call execution graph" has been introduced and proved to be deterministic.

The area of current research is the cloud environment for education and scientific computations. The Sisal implementation here described is not flexible enough because it requires the installation and .NET compiler. In our concept, the cloud interface gives transparent ability to execute programs in an arbitrary environment. JavaScript client do not demand installation, so small educational programs can be executed on client workstations. V8 server allows the language parser and some optimizations to be used at both (client and server) sides. The main aim for today is to make the language more available for users.

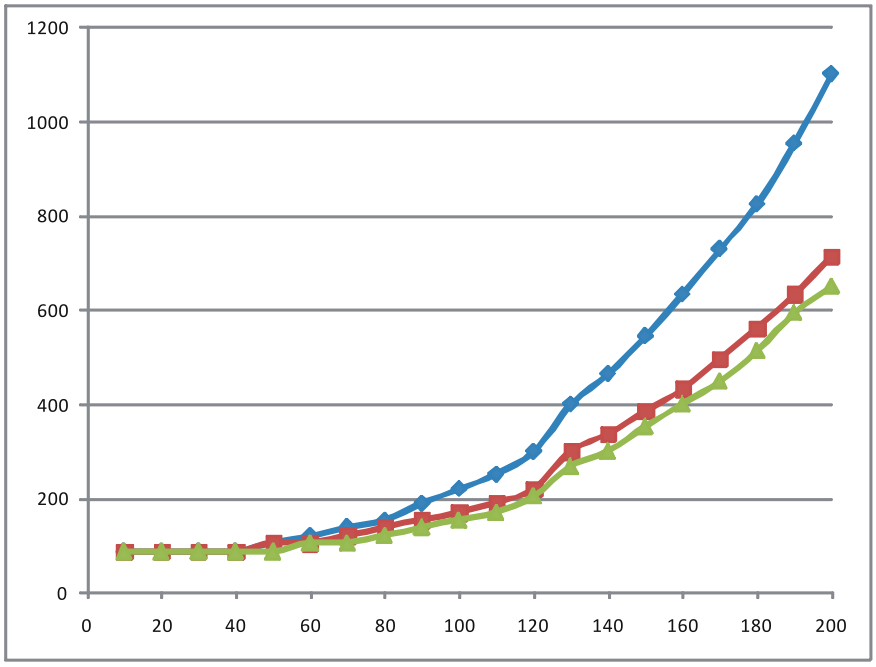

Figure 12. Smooth memory amount: X is the array size, Y is the memory in Mb; the lower curve (triangle markers) represents the optimized version with interprocedural analysis; the middle curve (square markers) represents the version optimized without interprocedural analysis

References

[1] McGraw J.R. et al. Sisal: Streams and iterations in a single assignment language, Language Reference Manual Version 1.1 / Lawrence Livermore Nat. Lab. Manual M-146. – Livermore, CA, 1983.

[2] Idrisov R. Sisal: Parallel Language Development // Proc. of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), May 30–31, 2012. – P. 38–42.

[3] Allen R., Kennedy K. Optimizing Compilers for Modern Architectures. – Morgan Kaufmann, 2002.

[4] Muchnik S. Compiler Design and Implementation. — Morgan Kaufmann, 1997.

[5] McGraw J.R. Sisal: Streams and Iterations in a Single Assignment Language, Language Reference Manual, Version 1.2. / McGraw J.R., Skedzielewski S.K., Allan S.J., Oldehoeft R.R., Glauert J., Kirkham C., Noyce B., Thomas R. – Livermore, 1985. – (Tech. Rep. / Lawrence Livermore National Laboratory; M-146, Rev. 1).

[6] Cann D.C. The Optimizing Sisal Compiler. – Livermore, 1992. – (Tech. Rep. / Lawrence Livermore National Laboratory; UCRL-MA-110080).

[7] Kasyanov V.N. SFP – An interactive visual environment for supporting of functional programming and supercomputing / Kasyanov V.N., Stasenko A.P., Gluhankov M.P., Dortman P.A., Pyjov K.A., Sinyakov A.I. // WSEAS Transactions on Computers. – Athens: WSEAS Press, 2006. – Vol. 5, N 9. – P. 2063–2070.

[8] Kasyanov V.N. Graph applications in programming // J. Programming and Computing Software. – 2001. – Vol. 27, Iss. 3. – P. 146–164.

[9] Karp R.M., Miller R.E. Parallel program schemata // J. of Computer and System Sciences. – 1969. – Vol. 3, Iss. 2. – P. 147–195.

[10] Kotov V.E., Narinyani A.S. On transformation of sequential programs into asynchronous parallel programs // Proc. IFIP Congress (1). – 1968. – P. 351–357.

[11] Karp R.M., Miller R.E. Properties of a model for parallel computations: determinacy, terminations, queueing // ASIAM J. of Applied Mathematics. – 1966. – Vol. 14. – P. 1390–1411.

[12] Adams D A. A Computation Model with Data Flow Sequencing. – 1968. – (Tech. Rep. / Computer Science Department, Stanford University, Stanford; CS-117).

[13] Rodriguez J.E. A Graph Model for Parallel Computations: PhD Thes. – Massachusetts Institute of Technology, Dept. of Electrical Engineering, Cambridge, Massachusetts, 1967.

[14] Kotov V.E. An algebra for parallelism based on Petri nets // Lect. Notes Comput. Sci. – 1978. – Vol. 64. – P. 39–55.

[15] Voevodin V.V., Voevodin Vl.V. Parallel Calculations. – St.Petersburg, 2002.

[16] Michie D. Memo functions and machine learning // Nature. – 1968. – N 218. – P. 19–22.