Dear Giancarlo,

The final copies of ALPUK92 were delivered to Springer at the beginning of the month. The person dealing with the manuscript has now come back to me and has asked for all authors to sign a copyright agreement before production can go ahead. It is only necessary for one of you to sign.

I enclose a copy for you to sign and return direct to Rosie Kemp at Springer (address below). Alternatively, Fax a copy to her. Any queries should be made to her until the second week of September as I am away until then.

I will also send an email copy to you if our mail system starts working again! It is Saturday however, and I don’t hold out much hope. In any event, a signed hard copy must be sent to Springer.

Regards,

Krysta Broda

Rosie Kemp
Springer Verlag
Springer House, 8 Alexandra Road
Wimbledon
London SW19 7JZ
Fax 081-947-4651
Telex 21531 SPRGB
Tel 081-947-5885

August 21st 1992

SEL Compiler & Abstract Analyzers

Giancarlo Succi, Giuseppe A. Mariño, Giancarlo Ollo, Diego Ob, Sergio Novella, Amedeo Pallo, Alexandre Regoli & Luca Viganò
DIB, Università di Genova
Genova, Italia

Abstract

This paper aims to evidence the potentialities of Prolog in the realization of a compiler and of some preprocessors for a subset equational language (SEL). Logic languages exhibit "implicit parallelism", which can be divided into "process parallelism" and "data parallelism". While most of the approaches are focused on the former, in this paper we analyze the latter: we present the architecture and instruction set of a virtual machine (the SAM) which aims to exploit the data parallelism present in programs written in SEL and thus to speed up the execution of SEL on standard architectures and to be the starting point of a data parallel implementation. The general structure of the SAM resembles that of the WAM, apart from not needing unification capabilities. Specialized instructions have been added in order to handle efficiently equational and subset assertions dealing with sets, and to point out to their data parallelism. The implementation of SEL is very similar to that of Prolog, since it exploits an abstract machine (SAM) which is very similar to the WAM of Prolog. The paper also contains a description of the strategies adopted during the realizations of all the phases of the compilation and of the abstract analyzers.

1 Introduction

Logic languages define explicitly the logic of programs, leaving implicit the control, and therefore they seem quite suited for an implicit parallel implementation. It is possible to approach this problem in different ways, but researchers have been focusing mostly on "process parallelism". This form of parallelism identification processes that may be executed in parallel and assign some of them to the available processors. And, Or- and Stream- parallelism belong to this class. Our aim is to exploit another possible approach, i.e., the "data parallel" one. Its philosophy consists of identifying clusters of data on which to apply the same operation. Hence the main point is to develop a new logic language or to introduce a proper data structure tailored to obtain a high degree of data parallelism. SEL, a logic language based on set data structure, seems to be well suited for this purpose. The parallel implementation of this language needs both efficient standard compilation from the high level language into a machine instruction language and an abstract analysis that identifies the data parallelism in the source program. The best choice for these tasks seems to be Prolog.
2 SEL: Subset Equational Language

This section is a brief introduction to the SEL and to the data parallel approach to logic programming; this section can be skipped by those already inside these topics. SEL, the Subset Equational Language, was developed by Jayaraman et al. [JN88] at UNC/Chapel Hill and at SUNY/Buffalo. The main feature of the language is the set data structure, because lots of people have experience from very different fields in representing problems as relation between sets. A SEL program consists of two kinds of assertions:

- equational assertion of the kind \( f(\text{terms}) = \text{expr} \), the function \( f \) is applied to the ground instances of \( \text{terms} \) and returns the ground instances of \( \text{expr} \);
- subset assertion of the kind \( f(\text{terms}) \text{ contains expr} \), the set function \( f \) is applied to the ground instances of \( \text{terms} \) and returns a new subset that is generated by the join of the ground instances of \( \text{expr} \).

Some examples of SEL programs can help to understand this approach.

\[
\text{square}(\text{X}) = \text{X}^2, \quad \text{squareSet}(\text{X}_1, \cdots, \text{X}_n) = \{ \text{X}_1^2, \cdots, \text{X}_n^2 \}.
\]

The first assertion computes the square of a given element, the second one computes the set of the squares of a given set. Some remarkable features are present in this simple situation, i.e., the multiple matching and the collect::all assumption.

- multiple matching: since no order is imposed over the elements of a set, a matching of the kind
  \[
  (\text{X}_1, \cdots, \text{X}_n)
  \]
  produces the matching of \( \text{X} \) all over the elements of the argument set.

- collect::all assumption: which states that the result of set function calls is the union of all subsets obtained by all the subset assertions matching the ground terms with every possible matching.

A data parallel implementation of SEL on a SIMD architecture can perform this operation in just one shot. It is necessary that the argument set has already been distributed among the available processors of the parallel machine. This task is performed at compile-time by a Prolog abstract analyzer that identifies not only a shared base of the set elements, but also the life time of these objects during the program execution. Moreover, it is possible to identify the size of the object involved in the parallel execution in order to save a large amount of memory.

In the same way it behaves the cartesian product of two sets:

\[
\text{cartProd}(\text{X}_1, \cdots, \text{X}_n) = \{ (\text{X}_1, \cdots, \text{X}_n) \}.
\]

However, more complicate patterns may be executed, such as:

\[
\text{perms}(\text{E}) = \{ (\text{E}) \}, \quad \text{perms}(\text{X}_1) \text{ contains distr(x, \text{perms}(\text{X}))},
\]

\[
distr(x, (\text{X}_1, \cdots, \text{X}_n)) \text{ contains } \{ (\text{X}_1) \}.
\]

This assertion computes all possible permutations of the elements of the set.

In order to obtain an efficient "data parallelism", it is necessary to generate all the sets matching the pattern in linear time and the application to all the sets of distr.

3 The SAM

In this chapter we will see the description of the abstract machine used in the implementation of SEL. As in Prolog, we can separate the execution of SEL programs into two different phases [BH89]:

- compilation, from SEL language to SAM assembler
- execution of the SAM code on the virtual SAM architecture and these correspond to two different phases of the implementation:
  - the development of a compiler targeted to the abstract machine
  - the implementation of the abstract machine on the real architecture.

The SAM belongs to the WAM [AK89] family, since its general structure resembles quite a lot that of the WAM, however it does not need full unification capabilities, therefore there is no need of the "tail" and faster store and match instructions replace the unify instructions. The SAM can be viewed as a wide of the SEL-WAM [Na88] from which it inherits most of the implementation strategies, which are extended with environment trimming, table of constants and the capability of harding function.

![Figure 1: General Structure of the SAM](image)

Figure 1 outlines the general structure of the SAM, in addition to the standard 4 components heap, stack, push-down list and processor, there is the
Active Memory: A memory whose cells both store data and perform computations. Its aim is to hold the sets so that data parallel operation can be executed on them.

![Active Memory Diagram](image)

Figure 2: The Active Memory

The structure of a cell is detailed in Figures 2 and 3: it is a multidimensional array of cells where each cell is composed of three elements: a processor, a set of registers and memory. The memory is organized into two parts: a stack, for performing local computations, and a region for keeping set elements, which are stored using hashing techniques: when there are some sets differing from one-another only by a few elements, it may be useful to store in the processing memory a superset of all of them, the base, and to represent each of the original sets (called remote sets) with a bit-vector (a 1 in its i-th position means that the i-th element of the base belongs to the remote set, and a 0 that it does not).

The SAM takes advantage of three main situations to exploit the data parallelism of problems, viz. when there are mappings of one set into another, when there are filters applied to a set, and when there are foldings of a set in a single element. When there is a map of one set into another, for instance in the set theoretic definition:

$$\text{set2} = \{f(x) : x \in \text{set1}\}$$

it is possible to compute set2 in one shot applying f to all the elements of set1 in parallel, as shown in figure 4.

Likewise, if there is a filter like:

$$\text{remote2} = \{x : x \in \text{remote1}, p(x)\}$$

remote2 can be computed in constant time (see Figure 5). Foldings are definitions of the kind:

$$f(\{\}) = k$$

$$f(\{x\}) = \{f(x)\}.$$  

Here it is possible to perform a tree-like computation in order to determine the result, as shown in figure 6, where z is applied to a and b, to c and d, to e and f, to g and h and to i and j at once, and then to each pair of results, and so on, until there is only one element left. Note that this class of operations is not deterministic, since no order is imposed on the elements of sets, e.g., the (pseudo) function nonDet:

$$\text{nonDet}(\{\}) = 0.$$  

$$\text{nonDet}(\{z\}) = \text{match}(z, \text{nonDet}(\{}).$$

applied to the set \{(2,3)\} can give 0, 2 or 4 as result, depending on which matching we choose, as shown in figure 5. However [MS89] demonstrated that if the folding function is commutative and associative the result is the same no matter of the matching; hence since plus is both associative and commutative, the result of applying det:

$$\text{det}(\{} = 0.$$  

$$\text{det}(\{z\}) = \text{plus}(\text{det}(\{}).$$

3.1 Data Objects

The type of data objects handled by the SAM are: atoms, integers, variables, boolean values, lists and sets. In the SEL language every term is represented by a word containing a value and a tag. The representation of lists or sets is obtained with a sequence of tagged pointer pairs, for the head and tail of them.
3.2 Data Areas

The code area, the Stack and the Heap, are the main data areas and hold all the compiled code.

- Stack is used to store, in an appropriate environment, all permanent variables associated with a clause. It is possible also to have a continuation performed by a continuation code pointer and a continuation environment pointer. For instance, when there are multiple subset rules, a choice point is created. It contains all information necessary to restore an earlier state of computation. This information consists of: pointer to the other subset rule or the branch pointers B1...Bn, all argument registers A1...An, the continuation program pointer CP, the current environment pointer CE, the last choice point LCP and the mode register M.

- Heap is used to store all structured objects. Anyway, when the structures are created on the Heap, they are not retracted like in the case of the Prolog WAM.

3.3 Registers

The registers used during an execution of a SEL program to store the current state are:

- P program pointer (to the code area)
- CP continuation pointer (to the code area)
- CE current environment (on the local stack)
- LCP last choice point (on the local stack)
- H top of heap
- S structure pointer (to the heap)
- CB current branch point register
- M mode register
- A1,A2,...An argument registers
- X1,X2,...Xn temporary registers
- B1,B2,...Bn branch pointers (to the code area)

As in the WAM, the A registers and X registers are identical, but they are used for different purposes: the first to pass the arguments of a function call, the others to hold the value of temporary variables.

3.4 SAM Instruction Set

SEL programs are compiled into instructions for an abstract machine (the SAM) similar to the WAM for Prolog. In general, each SEL symbol corresponds to one instruction. The entire instruction set can be divided into a number of classes. We describe compilation of SEL programs by describing each class of instructions.

- The go instructions correspond to the terms in the head of a rule, and are responsible for matching the rule against the argument of the function call, which are in the A registers.
- The storeIndirect instructions are used to return the result of the function call, and are generated for the last argument of a function.
- The match instructions are used to match arguments of lists or sets in the head of a rule.
- The put instructions are used to load the registers with the arguments before a function call is made.
- The store instructions are used to load arguments of lists and sets.
- The procedural instructions are used for control transfer, function invocation and environment allocation. They are:
The allocate instruction appears at the beginning of any equality rule that has two function calls in its body (in its flattened form - see section 4), or any subset rule that has one function call.

The execute instruction is used to call the last function in the body of any equality rule.

The proceed instruction ends an equality rule with no function calls in its body. All functions in the body of a subset rule and all functions in an equality rule except the last one are invoked with a call instruction.

The save.choice.point? instruction appears after the head and before the body of each subset rule.

The try.equ.also instruction appears before every equality rule which has another equality rule, with the same type of first argument, following it.

The try.sub.and instruction precedes the definition of every rule which has a subset rule, with the same type of first and the subset rules after. In both cases the argument is the code pointer for the next rule.

The switch.on.ground.term LT,L1,Ls is used to do clause indexing. This instruction appears at the beginning of any function that is defined with multiple rules having different first argument. Let L, L1, Ls be the addresses of the definitions which have a constant, a list or a set as their first argument respectively.

The mapping instructions are used to manage the operations which iterate over the elements of a set having the shape of generating a new set whose elements are functions of the elements of the original one; these mappings can be divided in three categories, depending on the space they need to perform the matching process (which is a multiple matching):

- Constant space
- Quadratic space
- Linear space

The insert instructions have a critical role for performing mappings and filters since they can act in parallel on all the AM cells of a set.

4 Implementing SEL Compiler

The first step of the compilation is flattening all expressions of the program. Flattening corresponds to transforming an expression into its dismembered form, a sequence of variable assignments only of the form \( X_i = f(X_{i1}, \ldots, X_{in}) \), to reflect leftmost innermost reduction order. For example, flattening:

\[
\text{append}([1], Y) = Y.
\]

\[
\text{append}([T], Y) = ([\text{append}(T, Y)].
\]

\[
\text{permutations}([1]) = \{1\}.
\]

\[
\text{permutations}([X,Y]) \text{ contains } \text{distr}(X, \text{permutations}(Y)).
\]

\[
\text{distr}([X,Y], z) \text{ contains } ([X,Y]).
\]

produces:

\[
\text{append}([1], Y) = Y.
\]

\[
\text{append}([T], Y) = [T] \text{ append}(T, Y) = T1.
\]

\[
\text{permutations}([1]) = \{1\}.
\]

\[
\text{permutations}([X,Y]) \text{ contains } V1 := \text{permutations}(Y) \text{ contains } V2, \text{ distr}(V1, V2) = V1
\]

and flattening

\[
f(x) = g(h(i(x))).
\]

becomes

\[
f(x) = z := i(x) = w, h(v) = Y, g(Y) = z.
\]

In the description that follows, we assume that we are dealing with a flattened SEL program. From the description of SEL it suffices that, in order to generate a SAM assembler code, we need recognize only two SEL assertions:

- equational assertion: in the form head :: body.
- subset assertion: in the form head contains body.

And, for any assertion, we must recognize only a few SEL patterns:

- atom: a string beginning with a low-case character or anything within 'anything'.
- variable: a string beginning with an upper-case character or with the character ':'.
- functor: in the form of: \text{identifier}(X_1, X_2, \ldots) where X's may be variables or atoms.
- list: in the form of:

\[
[ H \ T ]
\]

\[
[ H \ ]
\]

\[
[ X \ E ]
\]

where H and T may be variables, atoms or functors.

- set: in the form of:
where \( H \) and \( T \) may be variables, atoms or functors.

Therefore the resulting Prolog program that can parse a SEL program is very simple. It involves only three logical modules:

- Flattener: that generates, by reading the SEL program, a SEL flattened program.
- Reader: that reads the SEL flattened program with the predefined Prolog predicate read and gives its output to next module.
- Prolog Grammar: that scans the result produced by the Reader and generates the parse-tree corresponding to the SEL flattened program.

As we have just said, in general each SEL symbol corresponds to a SAM assembler instruction. Then, after arriving at a leaf of the parsing tree generated by the grammar, we can immediately generate the SAM assembler instruction corresponding to the Token recognized.

4.1 A Little Example

Now, as a further explanation of concepts just expressed, is presented the SAM compilation of a little fragment of SEL program:

\[
\text{function( H, T) = \{ H ! T \}}
\]

It is assumed to be the body of an equational assertion, for example:

\[
\text{body_list(A,F)} \rightarrow \text{b_open_sq(A,B)},
\]

\[
b\text{_head(B,C)},
b\text{_tail(C,D)},
b\text{_close_sq(E,F)}
\]

\[
b\text{_open_sq([A|B],B)} \rightarrow \{ A \rightarrow '(',
\]

\[
gem\_cod(\text{store\_val}\_\text{u}\_\text{e}(A))\}.
\]

\[
b\text{_head}(A,C) \rightarrow \text{atomo}(A,B),
\]

\[
is\_\text{mid}(A,B),
gem\_cod(\text{store\_val}\_\text{u}\_\text{e}(A))\}
\]

\[
b\text{_close_sq(E,F)}
\]

The predicate \text{gem\_cod/1} simply writes on the output file the SAM assembler instruction corresponding to its argument.

In our example, SAM assembler code produced is:

\[
\text{STORE\_IMD\_VAL}\_U\_E(A),
\]

\[
\text{STORE\_VAL}\_U\_E(V1),
\]

\[
\text{STORE\_VAL}\_U\_E(V2)
\]

where \( A \) is the Argument register in which the result has to be returned, \( V1 \) and \( V2 \) are two temporary registers into which atoms \( h \) and \( t \) were previously loaded.

4.2 Using Prolog: Reasons and Gains

The reasons for the choice of Prolog to create a compiler for Subset-Equational programs are substantially these:

- Using DOG Prolog Grammars makes the generation of the parsing trees of a program very easy. In fact, to obtain an efficient and clear parser, we have only to specify, using Prolog formalism, the complete SEL language grammar. Furthermore, as subproducts (almost free of charge) of parsing tree generation, we can obtain:
  - \text{Consistency Checking}: a SEL program, which satisfies the grammar, is sure to be syntactically correct.
  - \text{Type Checking}: during parsing it is easy to verify Type Consistency of program's rules.
  - \text{Error Management}: during parsing it is also always possible to find out errors and to give some useful diagnostic informations to the users.

- \text{SAM code generation} can be bound to the leaves of the parsing tree as \text{Semantic Actions} during the expansion of the tree itself.

- \text{Compiler Maintenance} will be easier than using any imperative language. Future changes in the Semantics of SEL language will involve only few modifications to the parser written in Prolog (e.g. some Prolog classes will be added and others will be deleted).
5 Abstract Analyzers

Abstract analyzers play an important role in the construction of an efficient implementation of a logic language. Their goal is to evidence relevant properties of the program, simply throughout its analysis and not its execution. With the help of these informations, the compiler will then be able to write a fast and concise code. The abstract analyzers discussed in this article are very relevant both for the sequential implementation and the parallel implementation on the SAM. The goals of our analyzers are:

- Persistence Analysis, that is composed of:
  - Shared Bases Analysis
  - Shared Structures Analysis
  - Destructive Update Analysis
- Object Size Analysis

6 Persistence Analyzes

Persistence analysis should offer a measurement of the lifetime of objects at compile-time. This problem is undecidable, and therefore we will focus our discussions on approximations through estimations, knowing that it is important not to underestimate the lifetime of objects, i.e. the analyses must always find all allocated objects. These analyses allow us to save process memory space, which is very relevant. Therefore our approach is highly set-oriented. We can distinguish three different kinds of analyses:

- Shared Bases Analysis, which allows us to adopt a common base in order to represent remote sets;
- Shared Structures Analysis, which allows us to find which objects can be shared among different structures;
- Destructive Update Analysis, which allows us to identify the possibility of destructing and simultaneously updating a set, instead of constructing and allocating a new one.

6.1 Shared Bases Analysis

The goal is find out which are the bases that are needed by a program, in order to be then able to determine which sets can share these bases. This analysis consists of three different phases:

a. Set-Flattening
b. Graph-Representation of connected components
c. Analysis of graph-connectivity

The first step consists of performing a flattening of the clauses, this means, as was seen before, that the clauses must be rewritten to a flattened form containing only ground terms and variables. Every non-ground term is to be substituted by a variable, which will then be unified to the non-ground term, and added to the body of the clause. E.g.:

\[
\text{set-cancel}([\text{a}]) = ([\text{a}])
\]

becomes (in a flattened form):

\[
\text{set-cancel}(2) = Z := [\text{a}], Y = [\text{a}]
\]

The second step consists of performing a set-flattening, i.e. the flattened clauses are rewritten in a set-flattened form which doesn't explicitly contain sets. The clause set-cancel will therefore become (in a set-flattened form):

\[
\text{set-cancel}(2) = Y := Z = [\text{a}], Y = [\text{a}]
\]

It is relevant to remark that even nested sets have to be set-flattened, e.g. an expression like \( g([\text{a}][\text{a}]), \) is transformed to \( g([\text{a}][\text{a}]), \) and the equalities \( W = [\text{a}][\text{a}], Z = [\text{a}] \) are added to the body of the clause. The final result of both flattening and set-flattening is to have all the non-ground terms and all the explicit sets handled by the clause at term-depth 1. Having obtained the set-flattened form of the clause, it is very simple to detect all the sets contained in the clause itself, as we just need to scan the body. Each set explicitly encountered in the body is associated to a node of the graph, and an arc will connect each couple of nodes sharing an element or a subset. The analysis of all the clauses will hence permit the construction of a graph corresponding to the SEL program. The graph is not totally connected, but it is formed by several groups of connected nodes, each node standing for an explicit set. We will associate a different base to each single group of nodes. Therefore each single set of a particular group will be represented on the corresponding shared base at compile-time. Since Prolog doesn't explicitly permit the management of graphs, we have represented the graph with the help of a list, whose single elements correspond to different nodes. In our program, however, we have chosen not to consider the area between nodes, but just to detect and represent which nodes are connected to which. Therefore we have adopted as a structure a list of lists, where each sub-list represents a set of connected nodes which are likely to share the same base.

Example:

\[
\text{functors} = \{ g/1 \}
\]

\[
\text{f}([x],[y],[z],[a]) \text{ contains } \{ x[y(g(v))]. \}
\]

Graph Structure : \([W1,W2,W3,W4,W5,W6],[W3]\)
ConnectedGroups : \([V1 = \{x\}, W2 = \{y\}, W4 = \{z\}, W5 = \{y\}, W6 = \{g(x)\}, \{W3 = \{z\}\}]\)
Out = \([\text{SuperBase}: [V1 = \{x\}, W2 = \{y\}, W4 = \{z\}, W5 = \{y\}, W6 = \{g(x)\}, \text{SuperBase}: [W3 = \{z\}]])\]
6.2 Shared Structures Analysis

The active memory of the SAM is strictly limited. The structure of the SAM, whose standard part derives from the WAM, identifies the sets of sets as a target of the shared structure. Aiming to save memory space, we should analyse it and when some sets can be shared on multiple sets, thus allowing them to adopt the same (shared) structure. As outlined before, this problem is undecidable, but we can implement and utilize an algorithm that will always overestimate the sharingness of a SEL variable, thus possibly inducing us to consider a non-shared variable as "shared". Anyway we can say that the algorithm will never fail, i.e. a shared variable will always be marked as "shared", and never as "non-shared".

The algorithm is very general: it can be applied to any shared structure, and not simply to sets. The strategy is to verify if an object may depend on the result: we analyze the flow of every object of the class. The algorithm will produce an output consisting of two lists: one list contains the shared objects, the other the non-shared ones. We are forced to handle the non-shared objects, in order to prevent infinite computations. This approach increases the speed of the algorithm, and, most of all, it does not negatively affect the successful result of computation: if an argument is non-shared, the corresponding branch of the computation will be cut, thus leading to the same result. Recursive calls must be handled very carefully, as they are likely to produce infinite loops. To prevent them, a table of the active computations is used, which allows us to test the arguments currently under analysis. This abstract analysis always comes to an end, since the number of classes of a SEL program is always finite.

The results collected in this analysis will serve as input of the DUA.

6.3 Destructive Update Analysis

In this analysis we examine the problem of the destructive update of the data structures. This requirement is again imposed by the bounds of the active memory of the SAM, we need to determine where and when it is possible to make the destructive update of a certain object. The strategy used is, like in the SSA, very general, but we specially concentrate our efforts on sets. The algorithm begins by analyzing each clause by itself, and it marks (on the base of some given rules) the objects that can be considered as potentially destructible. The recursive calls are treated as in the SSA. The management of the branch-points must be very careful: in multiple assertions, the objects may be reused in successive backtracks and a destructive update on these objects would be very harmful. The final output will provide a list of all the objects that may be destructively updated, in a safe and concise manner. This is a very difficult problem: we are presently working on a rough implementation of this algorithm.

Object Size Analysis

Assuming an execution on a parallel machine, we need to know the size of the objects of the program, in order to properly allocate the sets space. In this analysis these dimensions have been expressed as function of th input size. This problem is again undecidable, and therefore we look again for a size-approximation through an algorithm based on statistics. This algorithm has been developed by Debra and Lin and Hermenegildo [DLH90], and it works on relational Prolog-like languages. We first analyse the equational assertions and then the subset assertions. The algorithm begins by set-flattening all the assertions of the SEL program, as in the SBA. We use this set-flattened form to build the Data Dependency Graph (DDG) that highlights the program control flow. We can obtain the dimension relations between the program objects by analyzing the DDG. From these relations we must resume some closed formulae (i.e. difference equations) for size computation. We can distinguish two cases for the construction of these difference equations: non-recursive clauses need only a standard analysis, whereas recursive clauses need a formula normalization. The results obtained by the equational assertions analysis will be utilized in the subset assertions analysis. There are again two different situations: multiple subset assertions with the same left-hand side, and implicit iterations on the arguments of a set. With the help of different algorithms, we will be able to produce two approximations of the results dimensions.

8 Conclusions, Related Work, Future Trends

In this paper we have described a new abstract machine for the execution of subset logic languages, the SAM. Its aim is to speed up the execution of SEL on standard architectures and to be the starting point of a data parallel implementation.

Presently we have almost completed the SAM implementation on a RISC Sun4 architecture, and we are in the process of developing a prototypic version of it on the Connection Machine. The first performance figures that we have obtained from the Sun implementation are encouraging both in absolute terms and in comparison with the SEL-WAM.

In the meanwhile we are building the basis for the forthcoming development of a debugger, and we are presently working on the realization of a graphical programming environment, capable of being transported to different systems.

References


