The SAM Instruction Set
Giancarlo Succi, Giuseppe A. Marino & Giancarlo Colla
DIST-Università di Genova
I-16145 Genova
Italy

Abstract

Declarative languages exhibit both data parallelism and process parallelism. While the latter has been deeply investigated few has been done on the former. This paper presents the instruction set of the SAM (Subset Abstract Machine), an abstract machine aimed to exploit it for SEL. First the general philosophy of the machine is presented and then its instruction set is discussed: the data parallel primitives are explained and a distributed pattern matching is introduced. Eventually a possible integration with process parallelism is outlined and some conclusions are drawn.

1 Introduction

Logic languages define explicitly the logic of programs, leaving implicit the control, therefore they seem quite suited for an implicit parallel implementation. Researchers have been trying to exploit it for many years, however the results are quite unsatisfactory. As a matter of the facts, they have been focusing mostly on "process parallelism", i.e., the form of parallelism which identifies the processes which can be executed in parallel and analogous (part of) them to the available processors. And, Or and Stream parallel implementations fall in this class. The drawbacks of this approach are that it does not exploit all the intrinsic parallelism of problems, as it will be deeply explained later, and its heavy cost in terms of both time and space due to the high need of communications, synchronizations and processes management.

The approach described in this paper, the data parallel one, is rather different: its philosophy consists of identifying clusters of data on which to apply the same operation in parallel. A similar idea has been already exploited in imperative languages with excellent results. Hence the point is whether logic languages have "enough" data parallelism to justify this approach and, if it is so, what kind of collection could be used.

SEL, Subset Equational Language, is a logic language based on the set data structure: using it as a target language seems to lead to an affirmative answer to the first question. This paper discusses an abstract data parallel machine for SEL, the SAM (Subset Abstract Machine). Its general structure has already been presented in [SM91], here the focus will be on its instruction set and on its implementation on the target (real and possibly parallel) architecture. The paper is organized as follows: section 2 is a brief introduction to SEL, which can be skipped by those already inside these topics, section 3 presents the SAM and outlines its architecture, section 4 is devoted to the SAM instruction set, section 5 outlines a possible integration with process parallelism and section 6 draws some conclusions and evidences open problems for future research.

2 SEL: Subset Equational Language

This section is a brief introduction to SEL and to the data parallel approach to logic programming; as mentioned in the introduction it can be skipped by those already inside these topics.

SEL, the Subset Equational Language has been developed by Jayaraman et al. [JN88] at UNC/Chapel Hill and SUNY/Buffalo. This language handles sets in a clean, neat and simple way. Choosing sets as the core collection has also the advantage that lots of people have experience from many different fields in representing problems as relations between sets.

A SEL program is a sequence of two kind of assertions:

- equational assertions of the kind \( f(\text{terms}) = \text{expression} \).
- subset assertions of the kind \( f(\text{terms}) \geq \text{expression} \).

The meaning of this assertions is:

- equational assertion: the function \( f \) applied to the ground instances of \( \text{terms} \) is equal to the corresponding ground instances of \( \text{expression} \);
- subset assertion: the function \( f \) applied to the ground instances of \( \text{terms} \) contains the corresponding ground instances of \( \text{expression} \).

The language incorporates the collect-all assumption for subset assertions, which states that the result of a function application to ground terms is the union of all the subsets obtained by all the subset assertions matching the ground terms with all the possible matching. We do not go into details here, and for a complete description of the language [Succi] can be consulted.

Some examples of SEL programs can help understanding this approach. The first program we examine is aimed to compute the sets of the squares of a given set.

\[
\text{squareSet}(\{x\}) \text{ contains } \{x^2\}.
\]

Here it is present a remarkable feature of SEL, i.e., the multiple matching: since no order is imposed over the elements of a set, a matching of the kind \( \{x\} \) produces the matching of \( x \) with all the elements of the argument set; therefore, by the collect all assumption, the result is the set containing the squares of all the elements.

A data parallel implementation on a SIMD architecture, for instance, can perform this operation in just one shot: if the argument set is already distributed among the processors what is needed is just to ask each processor to square the element stored in it (and this can be done in parallel) plus some extra (constant time) bookkeeping.

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

\[
\text{cartProd}(\{x\}, \{y\}) \text{ contains } \{x,y\}.
\]

Here we have two nested set mappings, but the general philosophy is the same, and so can be the implementation. Also more complicate patterns can be handled in this way, like:

\[
\begin{align*}
\text{perms}([1]) & = \{(1)\}, \\
\text{perms}([x]) & \text{ contains } \text{distr}(x, \text{perms}(x)), \\
\text{distr}(x, \{x\}) & \text{ contains } \{x\}.
\end{align*}
\]

which determines all the possible permutations of the elements of a set. In this case the computation proceeds first generating all the sets matching the pattern in linear time (assuming to have enough available processors, otherwise we need some sort of virtualization) and then applying to all the sets distr.

Filters can be implemented with this approach too:

\[
\text{filter}(\{x\}) \text{ contains if } p(x) \text{ then } \{x\} \text{ else } \{\}.
\]

The function filter selects the elements of its argument set that satisfy the predicate \( p \); again we can have a data parallel implementation in just one shot, provided that we have enough processors.
The implementation of SEL is divided in two phases [IN88]:
- the development of a compiler targeted to an abstract machine,
- the implementation of the abstract machine on the real architecture.

The abstract machine is called SAM, Subset Abstract Machine. It belongs to the WAM [AK90] 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 "trail" and faster store and match instructions replace the unify ones. The SAM can be viewed as a sister 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 handling functions.

![Diagram](image1.png)

Figure 1: General Structure of the SAM

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,1 viz. 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.

![Diagram](image2.png)

Figure 2: The Active Memory

Figures 2 and 3 detail the structure of the AM: it is a multidimensional array of cells where each cell is composed by three elements: a processor, a set of registers and memory. The memory is organized in two parts: a stack, for performing local computations, and a region for keeping set elements, which are stored using sharing techniques: when there are some sets differing one-another only for 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 bitvector (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:

\[ set2 = \{ f(x) : x \in set1 \} \]

it is possible to compute set2 in one shot applying f to all the elements of set1 in parallel. Likewise if there is a filter like:

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

remote2 can be computed in constant time. Foldings are definitions of the kind:

\[ f(\emptyset) = x. \]
\[ f(\{x\}) = x(f(x)) \]

Here it is possible to perform a tree-like computation in order to determine the result, as is shown in figure 6, where x 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 last class of operations is not deterministic, since no order is imposed on the elements of sets, e.g., the (pseudo) function nonDet:

\[ \text{nonDet}(\{\}) = 0, \]
\[ \text{nonDet}(\{x\}) = \text{minus}(x, \text{nonDet}(x)). \]

![Diagram](image3.png)

Figure 3: Structure of a Cell

\[ set2 = \{ f(x) : x \in set1 \} \]

Figure 4: set2 is obtained mapping set1 through f

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

Figure 5: remote2 is computed filtering remote1 with p
applied to the set \{1,2,3\} can give 0, 2 or 4 as result, depending on which matching we choose. 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 set:

\[
\text{set}(\{1\}) = 0
\]

\[
\text{set}(\{a,t\}) = \text{plus}(x, \text{set}(t))
\]
to \{1,2,3\} is always 6.

4 Data Parallel Instruction Set

The instruction set of the SAM has been designed along two main guidelines: obviously the first is exploiting the data parallel construct of the language and the second is trying to maintain the global design as simple as possible in order to ease the implementation of the abstract machine on different kinds of architectures. While the instructions devoted to equational assertions has changed only slightly from the one of the SEL-WAM [Hay91], the one concerning subset assertions has been entirely revised.

4.1 Instruction Set for Equational Assertions

As we said title has been changed for equational assertions from the SEL-WAM, a minor change has been introduced to speed up the operation of choosing the (equational) clause to apply: it has been devised a kind of clause indexing strategy based on the analysis of not only the structure of the first argument but also of all the other arguments and therefore we designed the instruction:

\[
\text{switch on ground expr <exp> var list set functor constant}
\]

which on the basis of the kind of \langle expr\rangle jumps to the proper label or fails.

A remarkable modification has been performed for equational assertions dealing with sets, trying to identify and to take advantage of the implicit data parallelism. In order to clarify the analysis, assertions can be divided in two classes:

- those generating a new set, usually of the kind:

\[
\text{f}(\{x\}) = \{h(x,t)|m(x,t)\}.
\]

- those producing a single element (which can be a set, but not necessarily) usually of the kind:

\[
\text{f}(\{x\}) = \text{h}(p(x,t),m(x,t)).
\]

Under certain circumstances simple analysis can produce an optimized SAM code for both classes. Consider, for instance, when the former has the shape:

\[
\text{f}(\{x\}) = \{h(x)\}.
\]

we can rewrite this pattern as

\[
\text{f}(\{x\}) \text{ contains } \{h(x)\}.
\]

which can be efficiently implemented as it is explained in the subset assertion section of this paper, since it is a mapping between two sets through the function \text{h}.

On the other side when the latter is of the kind:

\[
\text{f}(\{\}) = x.
\]

\[
\text{f}(\{\}) \text{ contains } \{h(x)\}.
\]

there is the opportunity for folding, as it was explained in section 3. For this purpose it has been designed the fold-like SAM\(^3\) instructions:

\[
\text{fold} \text{.} \text{f} x z a z r h / 3
\]

\[
\text{fold} \text{.} \text{f} x z a z r h / 3
\]

which perform the computation on the set \text{Za} storing the result in \text{Zr} using as zero \text{Za}; the difference between the two is that in \text{fold} \text{.} f/2 is the identity function.

4.2 Instruction Set for Subset Assertions

The compilation of subset assertions for the SAM differs from the one targeted to the SEL-WAM since the SAM is aimed to set up a "nice" environment for "simple" porting on (data) parallel machines.

The first difference concerns the \text{collect} instruction which is substituted by \text{union}, \text{fission}, \text{insert}, and \text{finitset}; it is not just a renaming since what they do is quite different:

\text{union} \text{Za} \text{Zb} takes the union of two sets storing the result in the first one, i.e.,

\[
\text{za} := \text{za} \cup \text{zb}
\]

and performs the duplicate check;

\text{fission} \text{za} \text{zb} \text{zf} is like \text{union} apart from the fact that the union is performed only if \text{zf} is true, i.e.,

\[
\text{zf then za} := \text{za} \cup \text{zb}
\]

\text{insert} \text{za} \text{zb} \text{zb} inserts the element \text{zb} in the set \text{za}, i.e.,

\[
\text{za} := \text{za} \cup \{\text{zb}\}
\]

performing the duplicate check;

\text{finitset} \text{za} \text{zb} \text{zf} is like \text{insert} apart that the insertion is performed only if \text{zf} is true, i.e.,

\[
\text{zf then za} := \text{za} \cup \{\text{zb}\}
\]

These four new instructions have a critical role for performing mappings and filters since they can set in parallel on all the AM cells of a set.

Regarding pattern matching, two can be its purpose inside subset assertions:

\(^3\text{Adopting the WAM convention of calling Ai the argument registers, Xi the temporary ones and Yi the permanent ones, furthermore when it is not known whether the registers are temporary or permanent they are called Zi.}\)
(i) to identify an element of a given set,
(ii) to iterate over the elements of a set.

It is possible to distinguish at compile time these two situations because in the first case we deal with a ground element, like in:

```c
lookfor(x[1..]) = true;
```
or an already matched variable, like:

```c
h(x[1..]) contains f(x).
```

while in the latter there is a free variable. For the first case we can use the instructions:

```c
match_set_element Zm Zn Zb
match_set_element_ren Zm Zn Zc Zb
```

which match the element Zm against the set Zn storing in Zb the boolean result of the matching and in Zc the remainder of the set (only for match_set_element_ren).

### 4.3 Mapping Instructions

Most of the operations which iterate over the elements of a set have the shape of generating a new set whose elements are functions of the elements of the original one: this is the standard definition of mapping which was introduced in section 3. The 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:** this is the case of patterns of the kind:

  ```c
  f(x[1..]) contains g(x).
  ```

  in which we obviously need only constant space since we just scan the set; for this case we use the machine instructions

  ```c
  map_over Zm Zn Zx Zm end
  end.map_over Zm Zx
  ```

  which have the following behavior:

  - `map_over Zm Zn Zx Zm end`: produces the (possibly parallel) analysis of the set pointed by Zm using Zn as index, storing in Zx the value pointed by Zm and jumping to end if the set is empty;

  - `end.map_over Zm Zx`: increments the scanner Zx and updates Zm consistently; if the increment succeeds (i.e., we have not yet examined all the set) the execution jumps back to start else it goes to the next instruction.

- **Linear space:** when we have patterns of the kind:

  ```c
  f(x[1..]) contains g(x,t).
  ```

  in general we need linear space, since we need to build n copies of the set, being n the cardinality of the set. The virtual machine instructions for handling this situation are:

  ```c
  map.generating_copy Zm Zn Zx Zm Zx Zx start
  ```

  which have the following behavior:

  - `map.generating_copy Zm Zn Zx Zm Zx Zx end`: here Zm is the argument set {x[t]}, Zn to x and Zx to t; Zm contains what is pointed by Zn. If the argument set is empty the execution jumps to the label end, else a copy of it is generated without an element and the execution goes to the next instruction.

  - `end.map.generating Zm Zn Zx Zm Zx Zx begin`: if we have not yet complete the analysis of the argument set, Zn and Zx are differently matched against {x[t]}, Zm is properly updated and the execution jumps in the label start, else we go to the next instruction. Zm is added as parameter to speed up the creation of the new pattern.

- **Linear space:** patterns of this latter case need only linear space for matching when the "remainder" of the set -t is not used in the answer since in this situation we can destructively update it. The instructions are:

  ```c
  map.overriding.copy Zm Zn Zx Zm Zx Zx start
  ```

  with the following behavior:

  - `map.overriding.copy Zm Zn Zx Zm Zx Zx end` is almost identical to map.generating.copy;

  - `end.map.overriding Zm Zn Zx Zm Zx Zx start` differs from end.map.generating only in the destructively modification of the set pointed by Zx.

Simple abstract answers can be used to determine when this couple of instructions can be used.

An example compilation chunk can help understanding this new design. Consider the following code for the set of the squares of a given set:

```c
squareSet{[x..]} contains {x*x}.
```

the corresponding SAM instructions are:

```c
squareSet/Z:
allocate
get.set A1 Y1
get.variable A2 Y2
map_over Y1 Y3 Y4 end
begin:
put.value Y4 A1
put.value Y4 A2
put.variable Y5 A3
call mult/3
insert Y5 Y3 Y4
end.map_over Y3 Y4 begin
end:
deduplicate
```

### 4.4 Filtering Instructions

Certain classes of subset assertions can be defined as filters, e.g., they select which elements of a given set belong to another one. Their role in the SAM philosophy was discussed in section 3. Their general form is:

```c
p(x[,t]) contains if g(x,t) then [g(x,t)] else []
```
We can define three kind of filterings in a similar way as we did for the mappings apart that here we need a 
booleam value telling us whether to store the computed element in the set and the usage of insert instead of 
insert. Therefore the compilation of this assertion:

\[ p([x]) \text{ contains if } q(x) \text{ then } (g(x)) \text{ else } \{ \}. \]

is the following:

\[
\begin{align*}
p/2: & \\
& \text{map_over } Z_3 Z_1 Z_2 \text{ end} \\
& \begin{align*}
& \text{put_value } Z_3 \text{ } A_1 \\
& \text{put_variable } Z_2 A_2 \\
& \text{call } q/2 \\
& \text{put_value } Z_3 \text{ } A_1 \\
& \text{put_variable } Z_2 A_2 \\
& \text{call } g/2 \\
& \text{insert } Z_3 Z_2 Z_1 \\
& \text{end.map_over } Z_1 Z_2 \text{ start} \\
\end{align*}
\end{align*}
\]

Note that assertions of the kind of:

\[ \text{intersect} ([x], [x]) \text{ contains } [x]. \]

have to be handled like filters, since the second matching poses a logic (and implicit) constraint on the fact that 
the element of the first set belongs to the result; the compilation of this assertion is therefore:

\[
\begin{align*}
\text{intersect/3: } & \\
& \begin{align*}
& \text{get.set } A_1 X_3 \\
& \text{get.set } A_2 X_4 \\
& \text{get.variable } A_3 X_5 \\
& \text{map_over } X_3 X_6 X_7 \text{ end} \\
& \begin{align*}
& \text{match.set.element } X_7 X_4 X_8 \\
& \text{insert } X_4 X_7 X_8 \\
& \text{end.map_over } X_7 X_6 \text{ begin} \\
& \text{proceed} \\
\end{align*}
\end{align*}
\end{align*}
\]

The last situation to consider occurs when there is the need to perform matching and iteration at once, e.g.

\[ \text{select.father}([\text{family}(\text{father},..),]) \text{ contains } [\text{father}]. \]

where there is an iteration over the set selecting the first argument of the functor only when the functor is 
\text{family}/. This situation could be reduced to the equivalent:

\[
\begin{align*}
& \text{select.father}([x]) \text{ contains if } (\text{is.family}(x)) \\
& \begin{align*}
& \text{then } \{ \text{get.father}(x) \} \text{ else } \{ \}. \\
& \text{is.family}(\text{family}(..)) = \text{true} \\
& \text{get.father}(\text{family}(\text{father},..)) = \text{father}. \\
\end{align*}
\end{align*}
\]

however, to optimize the execution we prefer to duplicate the \text{map} family introducing:

\[
\begin{align*}
& \text{map_over.matching} \\
& \text{end.map_over.matching} \\
& \text{map.generating.matching} \\
& \text{end.map.generating.matching} \\
& \text{map.overriding.matching} \\
& \text{end.map.overriding.matching} \\
\end{align*}
\]

which have the extra argument specifying the pattern to be matched against replacing that of the pointed element. 
The problem is now how to efficiently perform pattern matching in this data parallel framework.

4.5 Distributed Pattern Matching

The purpose of pattern matching is to identify portion(s) of a structure comparing it against a template. 
The WAM uses the PD-list to accomplish this task, however this approach is intrinsically sequential, since the 
list is a FIFO structure. Had this approach been taken also for the SAM, there would have been the need of 
sequentializing the computation each time a pattern matching would have been required on the elements of a set, 
consequently creating a critical bottleneck. A completely different mechanism has been therefore devised.

Whenever there is the need of performing a pattern matching on the elements of a set, a template of the pattern 
is created in the local stack of each AM cell of the set. This template contains a reference to a free register of the 
cell for each of its free variables. The operation of matching can then be performed locally in the cell operating 
on the local set element and on the local template and at the end of it either a local fail flag is set or the registers 
contains the desired values.

![Figure 7: Local stack during the distributed pattern matching of the example.](image)

The operations for creating the template are:

\[
\begin{align*}
& \text{start.match } Z_a \\
& \text{store.temp.variable } Z_a \\
& \text{store.temp.value } Z_a \\
& \text{store.temp.functor } f/n \\
& \text{store.temp.set } Z_a \\
& \text{store.temp.list} \\
& \text{store.temp.constant } c \\
& \text{store.temp.unref} \\
\end{align*}
\]

\text{start.match } Z_a \text{ places in the register } Z_a \text{ the address of the top of the stack, which is the place where the matching } 
\text{process will start. store.temp.variable } Z_a \text{ stores in the stack a reference to the register } Z_a, \text{ where the result of } 
\text{the pattern matching will be placed, store.temp.value } Z_a \text{ store in the stack a reference to the register } Z_a \text{ which } 
\text{was already referred to by another store.temp.variable } Z_a \text{ in order to handle nested matching (the situation is}
5 Integration of Process Parallelism

The fact that this approach is based on data parallelism and not process parallelism does not mean that this paper claims that process parallelism shouldn't be exploited at all. It would be very interesting to try to couple some limited forms of process parallelism with data parallelism in the framework of SEL. Furthermore SEL seems to be a good candidate for such an integration since it seems quite simple to have a conservative or parallelism, parallelizing the execution of multiple subset assertor matching the same head. The design of the SAM does not require big changes to handle this extended framework: the communications and synchronizations can be confined in the try..on..then and in the naison instructions. There is a starting project in this field and there is the hope of getting some early figures by the beginning of 1992.

6 Conclusion

In this paper it is described the instruction set of the abstract machine we are developing for the data parallel execution of logic languages, the SAM. Presently it has been completed the implementation of the SAM on a Risc Sunf architecture and it is benchmarked. The very first figures seems promising. In the meanwhile the Connection Machine implementation has started and it is in the early stage the mentioned project for the process parallelism integration. Many are the open problems, such as the ones about the best object allocation scheme for sets and the ones about abstract analysers.

Acknowledgments

The first authors thanks Prof. Jajaraman, since most of the ideas of this paper are due to his outstanding supervision at SUNY/Buffalo.

References


