[SM91a] G. Succi and G. A. Marino. Abstract Analyzers for Data Parallel Declarative Languages. Technical report, DIST - Università di Genova, 1991.

[SM91b] G. Succi and G. A. Marino. Data Parallelism in Logic Programming. In *Proceedings of the ILPS'91 Pre Conference Workshop*, Paris, France, June 1991. Springer-Verlag – LNCS.

[SM92] G. Succi and G. A. Marino. The Design of an Abstract Machine for Subset Equational Languages. In *Proceedings of the 1992 European Wrkshop on Parallel Computing*, Barcellona, Spain, March 1992.

[SMC+91] G. Succi, G. A. Marino, G. Colla, D. Co, S. Novella, A. Pata, A. Regoli, and L. Viganò. SEL Compiler and Abstract Analyzers. In *Proceedings of the ALPUK'92*. Springer-Verlag – LNCS, 1991.

[Suc91] G. Succi. Set Representations in a Subset-Equational Language, February 1991. State University of New York at Buffalo – Master Thesis.

[Suc92] G. Succi. Exploiting Implicit Parallelismof Logic Languages with the SAM. In *Proceedings of the 1992 Symposiumon Applied Computing*, Kansas City, Kansas, March 1992.

[Tic89] E. Tick. A Performance Comparison of AND- and OR-Parallel Logic Programming architectures. In Levi and Martelli [LM89], pages 452–467.

[Tur79] D. A. Turner. A newimplementatioon technique for applicative languages. *Software Practice and Experience*, (9), 1979.

# 9 Conclusions

Several are the issues still open in this research. Abstract analyzers and garbage collectors are under s and development. A different way of mapping the Active Memory on the hypercube by means of hash tables is almost completed. Network optimizations and compilation enhancements are also hot points.

# Acknowledgments

This work has been partly supported by the italian ministery of university and scientific research (40% foundings). Most of the ideas presented in this paper has been discussed with B. Jayaraman. The authors thank researchers of DII/Parma for their support in the usage of the Connection Machine.

# References

[AK90] H. Ät-Kaci. *The WAM: A (Real) Tutorial*. Digital - Paris Research Laboratory, January 1990.

[BG89] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In Levi and Martelli [LM89], pages 471–486.

[BP92] A. K. Bansal and Potter. An Associative Model to Minimize Matching and Backtracking Overhead in Logic Programs with Large Knowledge Bases. *Engineering Applications in Artificial Intelligence*, 5(3), 1992.

[BPV92] A. K. Bansal, J. L. Potter, and Prasad L. V. Data-parallel Compilation and Query Power Extension of Large Kowledge Bases. In *Proceedings of the 1992 IEEE International Conference on Tools with AI*, 1992.

[Bru82] M. Bruynooghe. The memory management of Prolog implementation. In *Logic Programming*, pages 83–98. ACM Press, 1982.

[DGL+79] R. B. K. Dewar, A. Grand, S. Liu, J. T. Schwartz, and E. Schonberg. Programming by Refinement, as Exemplified by the SETL Representation Sublanguage. *ACM Transactions on Programming Languages and Systems*, 1(1):28–49, July 1979.

[DOPR91] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Logic Programming Language with Finite Sets. In *Proceedings of the 8th International Conference of Logic Programming*, Paris, France, 1991. MIT Press.

[Hil85] D. W. Hillis. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.

[Jay90a] B. Jayaraman. Broader Forms of Logic Programming. In *Proceedings of the Joint US-Japanese Workshop on Parallel Knowledge Systems and Logic Programming*, 1990.

[Jay90b] B. Jayaraman. Subset-Equational Programming in Intelligent Decision Systems. *Journal of Computers and Mathematics with Applications*, 17:133–146, 1990.

[Jay91] B. Jayaraman. Implementation of Subset-Equational Programs. *The Journal Of Logic Programming*, 1991.

[JN88] B. Jayaraman and A. Nair. Subset-Logic Programming: Application and Implementation. In *5th International Logic Programming Conference*, Seattle, August 1988.

[LM89] Giorgio Levi and Maurizio Martelli, editors. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, 1989. The MIT Press.

[Mel82] C. S. Mellish. An alternative to structure sharing in the implementation of a Prolog interpret In *Logic Programming*, pages 99–106. ACM Press, 1982.

[MS89] G. Marino and G. Succi. Data Structure for the Parallel Execution of Functional Languages. In G. Gries and J. Hartmanis, editors, *PARLE'89*, pages 346–356, Eindhoven, The Netherlands, June 1989. Springer-Verlag.

[Nai88] A. Nair. Compilation of Subset-Logic Programs, December 1988. University of North Carolina at Chapel Hill – Master Thesis.

[SC92] G. Succi and G. Colla. Checking for Duplicates in the CM2. Technical report, DIST- Università di Genova, March 1992.

Machine 2 with 8K processors with a Sun 4 front-end. Two kinds of results are reported for the CM-SAM one taking into account also the time for checking for duplicates and one without considering it. Th reason for such choice is that the structure of the CM-SAM often make easy the process of detecting when checking for duplicates can be avoided.

The SEL version of the tests is:

```
incrMap({X|_}) contains {X+1}.

evenFilter({X|_}) contains if (even(X)) {X} else {}.

sumFold({}) = 0.
sumFold({X|T}) = X + sumFold(T).
```

While the PROLOG one is:

```
incrMap([],[]).
incrMap([X|T],[Y|W]) :- Y is X+1, incrMap(T,W).

evenFilter([X|T],[X|W]) :- even(X),!, evenFilter(T,W).
evenFilter([_|T],W) :- evenFilter(T,W).

sumFold([],0).
sumFold([X|T],W) :- sumFold(T,Z), W is X + Z.
```

The following table summarizes the results.

| Program | Size | SICStus | SAM | CM-SAM | |
|---|---|---|---|---|---|
| | | | | with DC | w/o DC |
| `incrMap` | 10 | 10 | 12 | 7 | 0.22 |
| `incrMap` | 100 | 180 | 180 | 50 | 0.22 |
| `incrMap` | 1000 | 1620 | 3300 | 600 | 0.22 |
| `incrMap` | 10000 | 15000 | 405017 | 5800 | 0.22 |
| `evenFilter` | 10 | 10 | 11 | 8 | 0.27 |
| `evenFilter` | 100 | 169 | 175 | 56 | 0.27 |
| `evenFilter` | 1000 | 1740 | 3280 | 620 | 0.27 |
| `evenFilter` | 10000 | 17300 | 406034 | 6 | 0.27 |
| `sumFold` | 10 | 21 | 14 | 1.22 | 1.22 |
| `sumFold` | 100 | 131 | 40 | 1.22 | 1.22 |
| `sumFold` | 1000 | 1010 | 450 | 1.22 | 1.22 |
| `sumFold` | 10000 | 11360 | 3900 | 1.22 | 1.22 |

This results are pretty encouraging. First of all it the SAM behaves pretty well in comparison wit SICStus, which is one of the best and fastest PROLOG currently available: the SAM challenges SICStus in `incrSet` and `evenFilter` for small and mediumsized sets (up to 1000 elements) and outperforms it in `sumFold` for any size. The problem connected with large sized sets is checking for duplicates; abstra analyzers are under development to determine when this operation is necessary [JN88, SC92]. Note that these performances are obtained without any *ad hoc* optimization, whereas SICStus is implemented with many peephole optimizations: there is much space left for going faster also in the sequential case.

On the Connection Machine checking for duplicates carries a linear cost. This explains the linear incre of time for the execution of `incrSet` and `evenFilter`, while no duplicate check is required for `sumFold`. However, as it is mentioned above, quite often duplicate check can be avoided, or at least deferred till end of the computation. Consider the definition of `severalMaps`:

```
severalMaps({X|_}) contains {f(g(h(i(j(k(l(m(X)))))))).
```

Here there is no need of performing the duplicate check operation until after the call of `f`; a deep analysis of this topic is described in [SC92]. For these reasons also the time required without checkin duplicates is presented. Note that it is almost constant for any assertion. This means that the struct of this machine exploit pretty well the SIMD architecture of the Connection Machine. No checking for duplicates is required for the `sumFold` assertion, since it produces a single element, rather than a set

```
 [1] allProducts/3:                    oneProducts/3:
 [2]        allocate                          allocate
 [3]        get_set A1 Y1                     get_variable A1 Y1
 [4]        get_variable A2 Y2                get_set A2 Y2
 [5]        get_variable A3 Y3                get_variable A3 Y3
 [6]        map_over Y1 Y4 Y5                 map_over Y2 Y4 Y5
 [7] begin:   put_value A1 Y5          begin:   put_value A1 Y1
 [8]          put_value A2 Y2                   put_value A2 Y5
 [9]          put_variable A3 Y6                put_variable A3 Y6
[10]          call oneProducts/3                call */3
[11]          insert Y3 Y6                      insert Y3 Y6
[12]        end_map_over Y4 Y5 begin          end_map_over Y4 Y5 begin
[13] end:   deallocate                end:   deallocate
[14]        proceed                           proceed
```

Figure 11: SAMCode for `oneProducts` and `allProducts`.

care of it. Figure 10 presents a situation where such parallel environment is used. The SAMcode for the clauses `oneProducts` and `allProducts` is in figure 11. The *allProduct* execution cannot be parallel, because a nested parallel assertion exists (the *oneProduct* assertion).

There is therefore the need of sequentializing the data parallelismon the first set and exploiting it onl the second set. The `end_map_over` of line `[12]` will restart the data parallelismon {1,2,3}. Consequently there is the need of providing a mechanismfor stopping a data parallel execution on a set to resume it later. Since this situation can be nested (e.g. , if `oneProduct` called an assertion with data para execution on another set) a stack of such information must be saved. The parallel environment is used for this purpose: a frame containing information about the set in use and which elements of it has already been analyzed is stored on the stack each time a mapping occurs inside the execution of another mapping and it is popped at the end of the mapping.

So at line `[6]` a parallel environment for {1,2,3} is placed on the stack, an element of it is selected in ord to start the sequenzialization (for instance 1) and it is marked as "already analyzed" in the environmer Any reference to `Y1` is nowa reference to 1. Therefore line `[7]` put a 1 in register `A1`. Then the standard flowis followed in lines from`[8]` throu `[11]`. In line `[12]` there is the end of the mapping. At this point the parallel environment is popped fromthe stack and it is analyzed to see if the processing has bee completed. This is not the case here, since elements 2 and 3 of {1,2,3} has not yet been processed. A newelement is then picked out fromthe parallel environment, for instance 3, and the environment is saved back on the stack. Any reference to `Y1` is nowreference to the number 3. Again lines `[8]` throu `[11]` are executed and line `[12]` is reached. The parallel environment is again popped fromthe stack, element 2 is selected and then the environment is placed back on the stack. The execution jumps back to line 7, where 2 is placed in `A1`. Then lines `[8]` throu `[11]` are executed and line `[12]` is reached. Nowall the set {1,2,3} has been processed therefore the environment can be popped fromthe stack, the original data parallelismon {1,2,3} can be resumed and lines `[13]` and `[14]` are executed. Then the execution goes back to `allProduct` which is then completed in the normal way.

The need for parallel environment is present not only when there are nested calls to clauses inside mappin but any time there is a nested mapping, therefore also in situation like the one of `cartProduct` which computed the cartesian product of two sets.

```
cartProduct({X|_},{Y|_}) contains {pair(X,Y)}.
```

# 8 Experimental Results

Some benchmarks have been performed. It is interesting to compare it with *SICStus PROLOG*, one of the best PROLOGimplementations, based on the WAM. The SICStus tests as well as the sequential SAM ones are executed on a RISCSun 4 running SunOS4.0.3, while the parallel SAMtests run on a Connection

```
[1] doubleIncrSet/2:
[2]          allocate
[3]          get_set A1 Y1
[4]          get_variable A2 Y2
[5]          map_over Y1 Y3 Y4 end
[6] start:     put_value A1 Y4
[7]            put_variable A2 Y5
[8]            call incr/2
[9]            put_value A1 Y5
[10]           put_variable A2 Y6
[11]           call incr/2
[12]           insert Y2 Y6
[13]         end_map_over Y3 Y4 start
[14] end:    deallocate
[15]         proceed
```

Figure 9: SAMcode for `doubleIncrSet`.

to entries in this table which provides all the information that are needed to completely identify a se the Active Memory. Each entry of the CM-Data-Array contains the following fields:

- *type*, which specifies whether a set is a permanent set, `PER`, or is an intermediate result, `DDT` (The reason for this name is explained in the section about mapping);
- *tag*, which is the tag used in the tag register of the cells of the set in the Active Memory to identifi which cells belong to a set;
- *register number*, which determines in which register of the cells the element of an intermediate set are stored, it is 0 for permanent sets.

The use of the CM-Data-Array has many advantages with respect to a straight usage of the tag number (and of the register number for temporary sets). First of all the design is much cleaner and it is possibl have a higher degree of modularization. Then there is a higher degree of flexibility of the whole structu since the code never addresses directly objects in the hypercube. Furthermore, there may be differen entries in the table pointing to the same set in the AM; this can be achieved when *ad hoc* abstract analyzers - which are under development now [SM91a] - determine that two sets are identical: then it is possible t allocate the set just once in the Active Memory and to refer to it by means of two different entries of the table. There is one more reason for the usage of the CM-Data-Array which includes all the previous ones: it is the way in which the mapping operations are performed. The next section is about this problem

# 7 The Parallel Environment

The Connection Machine is a SIMD machine, therefore only one instruction at a time can be executed on all the processors. This peculiarity needs some special care, since there are data parallel flows requ MIMD structures. If a single operation must be performed on two sets whose elements are allocated in different active cells, then the execution may be parallel only on one set while a sequential execution m be performed on the other. A mechanism, called *parallel enviroment*, has been devised in order to take

```
oneProducts(X,{Y|_}) contains {X*Y}.
allProducts({X|_},S) contains oneProducts(X,S).
```

Figure 10: The Need of Parallel Environments.

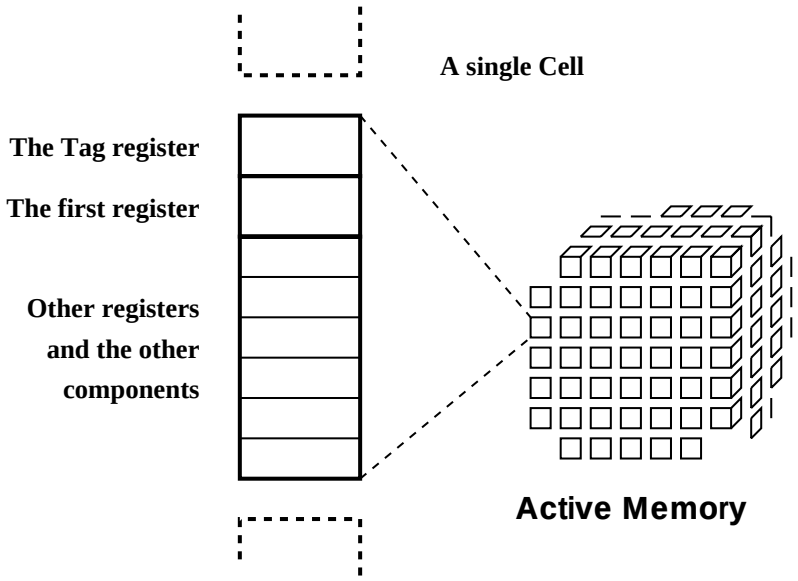


Figure 7: A Portion of the Active Memory.

Figure 7 illustrates the structure of the AM: it is the usual array of cells; in this implementation a reg of each cell, the *tag register*, is used to identify a permanent set. Other sets' elements can be stored in t cell, but they belong to temporary sets. The element of the permanent set is stored in the *first register* of the cell, therefore a permanent set in the AM is represented as a collection containing what is in the fi registers of the cells that have the same value in the tag register. Therefore a permanent set is identi just by a tag.

The elements of temporary sets are placed in the other registers of the cells, in the second, in the th and so on. Therefore a temporary set is identified by two numers: a tag plus a number, specifying in which registers of the cells pertaining to that temporary set its elements are stored.

While set does not contain duplicated elements, temporary set may contain them, because the duplicate check is performed only when the final set is created. Such design is due to the fact that the extra computation performed here are in data, a complete description of the problem can be found in [SC92].

# 6 The CM-Data-Array as an Interface between the Front-End and the CM2

The CM-Data-Array is a data structure stored in the front-end that is used by the main processor to address the sets stored in the Active Memory. It is a table (figure 8) and (in the present version of th CM-SAM) each entry of it identifies a set. Therefore any reference to sets in the front end are reference

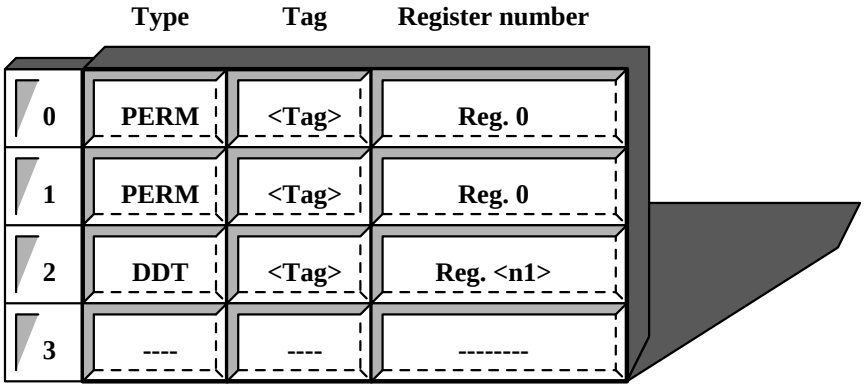


Figure 8: Structure of the CM-Data-Array.

```
[1] map_over
[2] put_value
[3] put_variable
[4] call incr/2
[5] put_value
[6] put_variable
[7] call incr/2
[8] end_map_over
```

Figure 6: Scheme of the SAMcode for `doubleIncrSet`

2. to rebuild entirely the mechanism of virtualization, tailoring it to this problem

3. to use a mixture of the first two. The first approach is classical in sequential implementations, howeve here the issue is more critical since the communication overhead is not under control and could lead t inefficiencies. The second is really hard to handle, since everything has to be taken care of, and there the risk (somehow paradoxical) that the difficulty of this task lead to a slow and buggy implementation. The best solution appears to be the third one which is a kind of compromise between (1) and (2) and it has been taken here. We have taken note that the operations on a set produce some intermediate data (on which a data parallel execution is possible). However migrating these new elements toward other cell is a useless operation because they are only transitory data. Therefore they are stored into the cells which they have origin. This handling minimizes the inter-processors communication into the CM2. Only the final set elements are copied into new cells leaving the routing mechanism up to the machine. As a matter of fact the elements belonging to sets may be separated in two classes:

   **(a)** those generated as intermediate results of a computation,

   **(b)** those produced as a final result of an assertion.

   To have a better understanding of this problem, consider the following assertions, where `doubleIncrSet` increments the elements of the argument set by 2.

   ```
   doubleIncrSet({X|_}) contains {incr(incr(X))}.
   incr(X) = X+1.
   ```

   Figure 6 illustrates the lines of the SAMcode generated for this assertion that executes the increment `X` in parallel on all the processors on which the set[1] is stored. Both `[4]` and `[7]` act on a whole set: they increment the value of each element stored inside the set. However there is a difference between the two situations, *viz.* while the result of `[4]` is needed only locally, the result of `[7]` is a global result which must be kept for further computation. In other words, the result of `[4]` pertains to class (a) while the result of `[7]` to class (b). The choice of this design is to store a set of class (a) in the same cells as original set and to store a set of class (b) in a new cluster of cells. Since the abstract design of the AM associates to each set element its own cell, this kind of implementation amounts to handling explicitely the virtualization for sets of class (a) and leaving this task to the underlying system for sets of class (b).

   Such a choice yields an efficient and fast way of generating intermediate sets, which are needed only locally and not globally, therefore they do not need to be burdened by communications. Furthermore this choice keeps a simple structure for global sets: their elements maintain the one to one correspondance with an AMcell, possibly a virtual one, making easy the task of bookkeeping the system, identifying global sets, dereferencing their elements and so on, as it is explained in the next section.

   The implementation of the AM on the Connection Machine (*CM-AM*) is built upon two major entities:

   (a) the Active Memory itself, which contains the sets elements in its cells,

   (b) the *CM Data-Array*, which is used to identify the various sets in the Active Memory.

[1] This is just a scheme of the real code: not all the SAM instructions needed are placed and the arguments are left over in order to keep the presentation simpler.

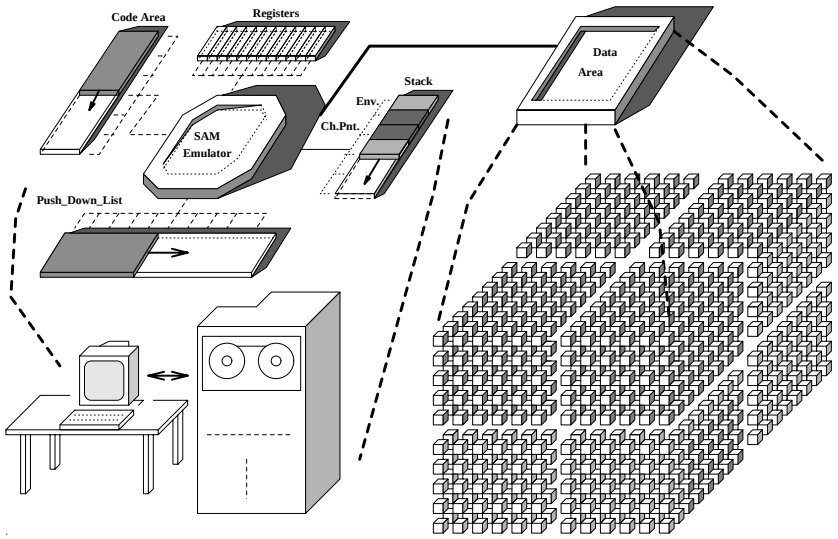


Figure 4: General Structure of the SAM

The **stack** handles the computation. Two different elements are stored onto it: **environments** and **choice points** [Bru82]. An environment is saved onto the stack when an assertion is called by another. It contains the informations (global registers, continuation pointer, environment address) needed to continue the execu when the called SAM subroutine is finished. A choice point is used to save the SAM status when a multiple defined subset assertion is evaluated.

The **Push_Down_List** is used to perform matching over the elements belonging to sets (such as $\{functor(X, a) \mid \_\}$). The structured data skeleton is built onto the PDL by some dedicated SAL instructions (*store_pdl*).

The **heap** contains functors and lists; figure 5 shows how these elements are stored into the heap. A list is characterized by a sequence of cell couples: the first contains the current element, while the second addresse remainder of the list. A functor having $n$-arity is characterized by e sequence of $n+1$ cells. The first conta the functor identifier and its arity, the last $n$ cells store its terms. Obviously these structured data ma nested each other.

# 5 Modelling the CM2 as an Active memory

The **active memory** is a collection of memory cells that have a computing power, i.e. they not only store data, but also perform a data parallel execution. Therefore, as be seen in figure 4, the *active memory* is implemented on the CM2 where a physical parallel execution is possible (while other machine modules, e.g. the *stack*, th *heap* and so on, resides on the front-end because their executing model is intrinsically sequential). Every si set data is bound to a single *active memory* cell that has a computing capability. The version of the CM2 that we use here has 16K processors and the set dimension of some source code may exceed this number. Therefore, it is necessary binding more cells to a single physical processors. This operation may be performed follov three different approaches:

1. to leave everything up to the machine,

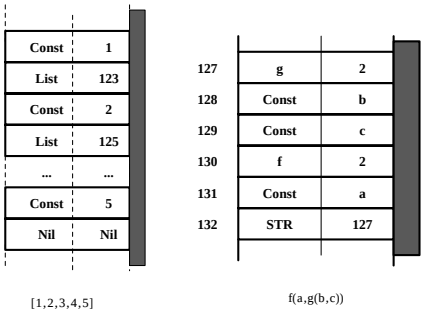


Figure 5: The Physical Implementation of lists and functors

```
[1]  allocate                    % An environment is
                                 % allocated onto the stack.
[2]  get_set A1 Y1               % It verifies if A1 contains
                                 % a set reference.
[3]  start_set_match Y2
[4]  store_pdl_functor func/2        % The set matching is
[5]  store_pdl_const a               % prepared by store_pdl
[6]  store_pdl_value Y3              % instructions.

[7]  get_variable A2 Y4          % Y4 addresses where writing
                                 % the solution.

[8]  map_over_matching Y1 Y5 Y2 end  % Iteration and matching on
                                     % set elements starts
[9]  start:  put_value A1 Y3         % Put instructions handle
[10]         put_variable A2 Y6      % value passing to the g/2
[11]         call g/2                % assertion.
[12]         union Y4 Y6             % The new solution is
                                     % collected to the preceding
                                     % ones.
[13] end_map_over_matching Y5 Y2 start
[14] end: deallocate                 % The environment is deallocated
                                     % from the stack.
[15]     proceed
```

Figure 3: An example of SEL assembly code

- a **Push_Down_List**,
- a **Code Area**,
- an **Emulator**,
- an **Active Memory**.

The figure puts into evidence that the abstract machine software acts partially on the CM2 front-end (in our implementation it is a SUN4) and partially on the data-parallel processors. The stack, the registers, the h the PDL and the code area handle the memory in the front-end, while the **active memory** handles memory and processors of the CM2. In fact the name *active memory* has been chosen because it changes the CM2 into a memory that is able to performcomputation. In the last part of this section the principal elements of tl SAMare outlined.

The registers used during an execution of a SEL programare:

- **PC** programcounter: it addresses the next instruction to be executed.
- **CP** continuation pointer: the PC value is saved into CP when a newnested assertion is called.
- **CE** current environment: it addresses the top environment onto the stack.
- **LCP** last choice point: it addresses the top choice point onto the stack (it is used when multiple definitio of a subset assertion exist).
- **H** it addresses the top of the heap.
- **S** structure pointer: the S register is used to performmatching operations on structured data.
- **A1, A2, . . . , An** argument registers: they are used to performargument passing between called and calling assertion.
- **X1, X2, . . . , Xn** temporary registers: they are used to store local vaariables.

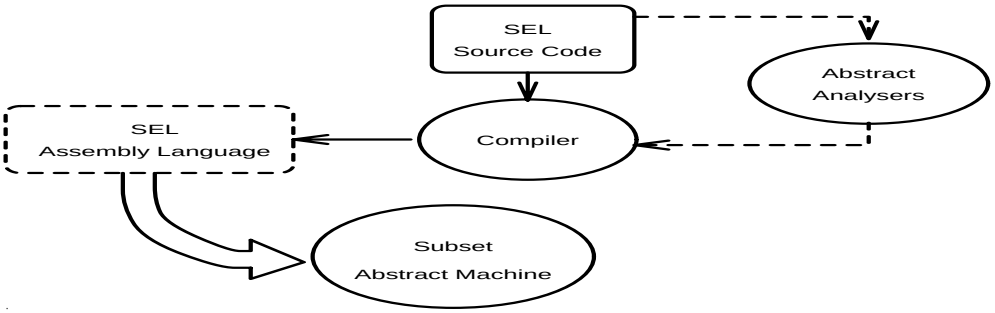


Figure 2: The SEL execution phases

The SEL compiler doesn't translate directly the SEL source code into CM2 machine code or, for example, into C*, because this approach is excessively difficult. Our compiler translates SEL into a sequence of instructi that are processed by an underlying software layer (the subset abstract machine); its goal is to approach t CM2 to the computing model of SEL (the next section presents the principal features of our abstract machine).

Figure 2 puts into evidence that a subphase of compilation may be managed by some abstract analyzers [Nai88]. Their goal is to evidence relevant properties of the program, simply throughout its analysis and not execution. With the help of these informations the compiler may write more fast and concise code speeding up the execution of the program. For example the compiler performs some optimizations, such as **LCO** (i.e. Last Call Optimization) and **Environment Trimming** ([AK90]). Besides, using a persistency analysis, it can have a measurement of the lifetime of objects at compile-time. A detailed description of abstract analyzers can found in [SMC+91].

Figure 3 presents the SAL code of the assertion

$$Chk(\{func(a, X) \mid \_\}) \ contains \ g(X)$$

This assertion first locates all the elements belonging to the set that are functors with identifier **func/2)** a first term literal **a**, then executes the *g/2* assertion on every matching second term(*X*). In line [1] of figure an environment is allocated onto the stack (the environment contains permanent variables and some abstract machine registers that are overwritten executing nested assertions). The *get_set* instruction of line [2] ve if the first argument of *Chk* (that is put into register A1) is a set. At this point the set matching operation prepared by storing some informations into the **Push_Down_List**. The register Y2 contains the address of first cell of the PDL where the matching skeleton is saved. Line [3], [4], [5] write respectively the functor identif first ground term and the variable to which the second term of the functor must be bound. Line [7] writes into register Y4 where writing the solution. From line [9] to line [13] the map_over instructions perform the itera over the elements belonging to the set. The matching operation is executed at the same time over the elements belonging to the set (that are distributed into the local memories of CM2 data-parallel processors) and produ a new collection of data. In line [9] and [10] the arguments of the *g/2* assertion are prepared; the register addresses the collection of data generated by the set matching and A2 points the environment cell where the solution must be put. When the assertion g/2 is executed, the abstract machine verifies that its first argumen contains a reference to data distributed over CM2 processors and, therefore, preforms a parallel computation them. The union instruction at line [12] collects the new solution to the preceding ones (*collect-all assumption*

# 4 The Subset Abstract Machine

The **SAM** belongs to the **WAM** [AK90] family, since its general structure resembles that of the WAM. However there are numerous differences. While the core of the WAM is the handling of unification capabilties, the SAM core is the handling of sets.

The Subset Abstract Machine is a software layer that adjusts the machine architecture to execute SEL assertions. In the introduction we have written that only programmers that know the architecture details parallel computers are able to achive their best performance. In our implementation the architectural deta are handled by the abstract machine: the SAM distributes data into the different processors of the CM2, the SAM performs parallel computation and transfers new data into free processors. In this way the the computing model of CM2 is completely hidden to the programmer.

Figure 4 outlines the general structure of the SAM; the most significant elements are:

- a collection of machine **registers**,
- a **Stack**,

*f(terms)* $\supseteq$ *expression*.

Their execution may be divided into two parts. In the first the matching of the arguments in the assertion head are verified. The operation replaces variables by ground terms. In the second part of the execution th **expression** side (see above) is solved. At this point the assertion *f* is equal or contains (in the case of sub assertion) the ground instances of *expression*.

The following example describes how we can define the permutation of the elements belonging to a set.

```
perms({}) = {[]}.
perms({x|t}) contains distr(x,perms(t)).
distr(x,{t|_}) contains {[x|t]}.
```

When a query (e.g. `?-perms{A,B,C}`) must be executed, the matching between the calling function and the arguments of the assertion head is verified. The behaviour of the set matching is rather peculiar. In the examp the matching between the ground term `{A,B,C}` and the set definition `{x|t}` produces different results.

- `x` $\leftarrow$ `A, T` $\leftarrow$ `{B,C}`
- `x` $\leftarrow$ `B, T` $\leftarrow$ `{A,C}`
- `x` $\leftarrow$ `C, T` $\leftarrow$ `{B,A}`

This behaviour is what we call *implicit iteration*, *viz.*, selecting the elements of a set one at a time. For ever possible matching the *distr* assertion is executed; its first argument is the ground term bound to the $x$ variabl and the second is the collection of every possible permutation applied on subset $t$ (they are obtained by t recursively execution of *perm*). For every different matching the *distr* assertion produces a new result (figu 1), these results are collected together and they become the final result of the assertion.

The implicit iteration on the elements belonging to a set is very useful to implement SEL on a data paralle architecture. In fact we can notice that the same operations are applied on different data. This implicit da parallelism may be exploited by distributing elements belonging to sets on the processors of a SIMD machine and performing parallel execution. The next examples performs some simple operations: the product, the instersection and the square of every element into the set.

```
setproduct({X|_},{Y|_}) contains {pair(X,Y)}.
intersect({X|_},{X|_}) contains {X}.
setsquare({X|_}) contains {X*X}.
```

**setproduct** produces every possible pairs of the elements belonging to the first and second set, while **intersect** produces the intersection between the two sets. **setsquare** computes a new set that is created from the square of the elements belonging to the starting set.

# 3 The SEL Compiler

This section describes the compiler. Like for Prolog, we can locate two different phases in SEL execution [JN8 (see figure 2):

- **compilation** from SEL to SEL assembler language (defined *SAL*),
- **execution** of SAL code on the **S**ubset **A**bstract **M**achine (defined *SAM*).

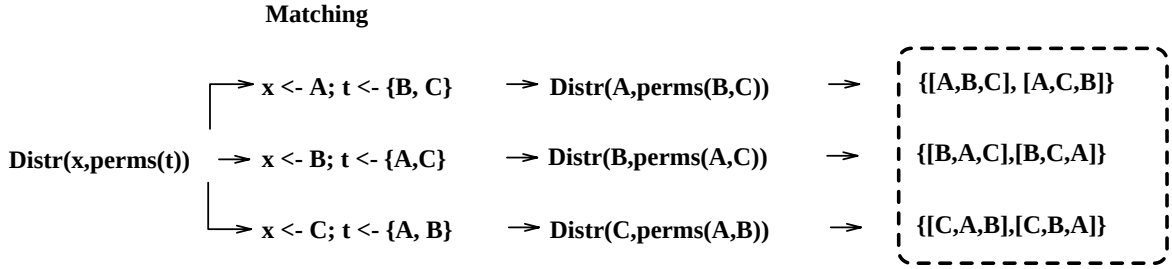


Figure 1: A skeleton of `perms` execution.

# Modelling the Connection Machine 2 as an Emulator of Subset-Based Declarative Languages

Giancarlo Succi, Giuseppe A. Marino, Giancarlo Colla

DIST-Università di Genova
via Opera Pia 11a, I-16145 Genova, Italia

**Abstract**

This paper describes a data-parallel implementation of the Subset equational language on the Connection Machine 2, the **S**ubset **A**bstract **M**achine. In the first part the difficulty of writing programs that really exploit the parallelism of the Connection Machine 2 is put into evidence. First of all, the programmer must know the architecture details of the parallel machine, besides he must chose an algorithm that is suited to data-parallelism (in the case of the CM2). The use of imperative languages is difficult because they greatly feel the effect of the above-mentioned problems. They lay complitely upon the programmer the responsibility of chosing the proper algorithm and of implementing it. A programmer that uses a declarative language describes what the problem really is and not the operation sequence needed to solve it. It is up to compiler and executor to divide the job into different processes and to distribute both them and data on machine processors. The SEL abstract machine implementation, that is described in the paper, hides to the user the architecture of the CM2 using the Connection Machine as an **Active Memory**. The last sections of the paper describe the principal modules that constitute the SAM.

## 1 Introduction

Nowadays we witness the development of data parallel and process parallel architectures. They have a great computational power, but it is difficult writing programs that really exploit it. The reasons that explain this situation are the following. First of all, only programmers that know architecture details of parallel computers are able to achieve the best performance from them. As a matter of fact it is very important how data and processes are developed and assigned to processors. A good programmer minimizes the execution overhead that inter-process communication and shared variables access may cause. Besides a good implementation is not sufficient to use at best a parallel architecture, because chosing the algorithms that exploit the machine parallelism very important too. Unfortunately the programming languages that are widely used are the imperative ones, they greatly feel the effect of the above-mentioned problems. These languages declare explicitly the operations that the processors have to do to reach the solution (i.e. it is defined the control of the program). Evidently this approach lays completely upon the programmer the responsibility of chosing the proper algorithm and of implementing it. We think that declarative languages are most suited to be used on parallel architectures because they define explicitly the logic of the program, leaving the control implicit. The "*only*" worry of the programmer is to describe what the problem really is and not the operation sequence needed to solve it. It is up to compiler and executor to devide the job into different processes and to distribute both them and data on machine processors. The number of logic and functional languages implemented on parallel architecture is growing. However most of them only exploit either data-parallelism or process-parallelism. We have chosen to implement a logical language based on the set data structure, the Subset Equational Language (called *SEL*). It is suited to be implemented both on a data parallel machine and on a process-parallel one. In this paper we describe the data-parallel implementation of SEL on a Connection Machine 2.

## 2 The SEL Language

The Subset Equational Language was developed by Jayaraman et al. [JN88] at UNC/Chapel Hill at SUNY/Buffalo. In this section we give only an overview of the language throught some examples, a more exhaustive description of it can be found in [Suc91].

A SEL program is a collection of equational and subset assertions:

*f(terms) = expression.*