CM2 as an active memory to implement declarative languages

GIANCARLO SUCCI, JOY MARINO and GIANCARLO COLLA
DIST-Università di Genova, via Opera Pia 11a, I-16145 Genova, Italy

Received August 1992

The implementation of the subset abstract machine active memory on the CM2 is described. The subset abstract machine (SAM) is a data parallel abstract machine used to implement the subset equational language (SEL). A milestone toward the full implementation of the SAM on the CM is the implementation of the active memory, viz. the part of the SAM which is used both to store sets and to perform (data) parallel operation on them. Several issues are raised by this implementation: virtualizing of the processors of the CM2, bookkeeping of the sets, managing the SIMD restrictions, and so on. These problems are explained and discussed, and some solutions are outlined. The solutions are tested against some examples. Some interesting early figures of execution times are presented.

Keywords: Declarative languages, SIMD architectures, data parallelism

1. Introduction

The current trend in the architecture of processors is towards new solutions, and programming them properly is becoming harder. Logic programming can be viewed as a good means for solving such problems: it does not explicitly specify the control flow of an algorithm; rather it defines the logic of an algorithm leaving the task of producing an efficient and effective executable code with respect to the target architecture to the compiler author. The price paid by such an approach is a possible loss in execution speed with respect to having specialized solutions for each problem, using an approach targeted to the specific hardware on which the program will run. However, there are many benefits in terms of simplicity, portability, modularity, reusability, and suitability to abstract analysers and verification tools. All these benefits can lead, in the real world, to a faster code. Furthermore, only the compiler author need know the details of the hardware of the target machine; this implies that the range of architectures that each programmer can use is much wider.

One of the consequences of taking such a hardware independent approach is that the same code can run both on sequential and parallel architecture. In the latter case, the compiler has to recognise what can run in parallel. In other words, the task of the compiler is to identify and to exploit (part or all of) the implicit parallelism present inside a program written in a logic language.

Two major forms of parallelism are present in logic languages: process parallelism and data
parallelism. Process parallelism is the form of parallelism which is obtained by parallelizing the execution of independent clauses [1–3]. Data parallelism identifies the clusters of data over which the same operation is applied [4, 5]. While most of the research has focussed on the former, this paper is devoted to the latter.

A data parallel implementation of a logic language (SEL) is described on a SIMD machine, the CM2. The equation, SIMD = data parallelism, is not always true\(^1\), but SIMD machines are good candidates for the efficient implementation of SEL. The standard approach of using an abstract machine as an intermediate structure for the implementation is used. The subset abstract machine (SAM) is the abstract machine developed for the data parallel implementation of SEL. The implementation of the core of the SAM, the active memory, on the connection machine 2 is first described. Then an overview of some background concepts about SEL, the CM2 and the SAM first are given. The problem of mapping the SAM active memory on the hypercube of the CM2 is discussed. The structure of the CM-AM is given and the way in which sets are managed by the main processor is presented. An example of the flow of computation is also given. Not all data parallelism implies SIMD execution; the way in which the problem is solved, by means of a mechanism called parallel environment, is also presented.

2. The SEL language and the SAM abstract machine

This is not a complete and exhaustive description of SEL, the CM2 and the SAM [7–10]. It is just a very brief overview to remind the reader of their major features.

The subset equational language (SEL) is a logic language based on two kinds of assertions: equational assertions and subset assertions. Equational assertions have the usual meaning. Subset assertions define a subset relationship between a \(lhs\) and a \(rhs\). This relation is so that any ground instance of \(rhs\) is a subset of the relative ground instance of \(lhs\). In case there is more than one clause with the same \(lhs\), the set defined by such \(lhs\) is the union of all the corresponding \(rhs\). An operation that is performed over a set is therefore executed all over the elements belonging to it [17]. This is an important feature of the SEL that is used to develop the parallel data implementation.

The CM2 is a SIMD machine. The version used has 16K processors organized in a hypercube architecture. The processors can be virtualized by means of a mechanism called a data vault. The routing strategy is that of the standard hypercube, with redirection to a random node in case of collision. The user accesses the machine through a front end which is a RISC Sun 4. There is also a high speed broadcasting bus connecting the front end with the processors.

The SAM is the abstract machine used in the implementation of SEL: at this time a sequential implementation a RISC Sun 4 is complete and a data parallel implementation for a connection machine 2 is at an advanced stage (some trivial benchmarks are available). Its structure is strictly related to the SEL-WAM [11, 12], the first abstract machine for SEL, and to the WAM [13, 19]. Like the SEL-WAM, it does not need unification capabilities and therefore there is no need for the trail in it. The structure of the SAM is presented in Fig. 1. Heap, stack, push-down list and processor have the same meaning as in the SEL-WAM, apart from the handling of sets.

\(^1\)The discussion on this topic is in the section about parallel environments.
Fig. 1. General structure of the SAM.

Fig. 2. Active memory topology.

The active memory is used to store sets: it is a multidimensional array of active cells (Fig. 2). Each active cell (Fig. 3) is a simple sequential processor that performs operations over a single element. It contains several registers, flags, a heap and a stack for local computations. The elements belonging to a set are distributed over a cluster of active cells. A group of active cells performs operations if it identifies the set over which the computation is executed.

3. Implementing the active memory on the CM2

Various approaches can be taken to implement the active memory on the CM2. The first choice to be made is the language. Despite the fact that C PARIS [9] allows deep usage of the CM2 hardware, C* has been selected since its simplicity overperforms its pitfalls. At this stage of the
implementation, the focus is placed on the global structure of the system rather than in *ad hoc* optimizations. Furthermore, it is always possible to port this version in C PARIS in a later stage of the development of the system, in order to provide peephole enhancements.

Then decisions must be made about virtualizing the processors. The version of the CM2 used has 16K processors. Therefore, since this is a limited amount of space, it may be that there is not enough space to store a set. The same also happens in regular sequential implementations when the memory is full of data; however there are two main differences in this situation: the space available is lower than that of a sequential implementation, and the problem regards processing cells and not memory cells. Sequential implementations of declarative languages often rely on the ability of the operating system in handling the virtual memory, sometimes provide means for garbage collection and hardly ever use abstract analysers to determine which data can be safely reused and/or destructively updated. Since this is the first experience of such an approach, it has been decided not to perform garbage collection nor abstract analysis yet. Garbage collection is important, it will therefore be addressed soon. Abstract analysers are already under development [14] and will be added to future versions.

The CM2 provides a means for virtualizing the usage of processors, therefore three paths can be taken:

1. To leave everything up to the machine;
2. To entirely rebuild the mechanism of virtualization, tailoring it to this problem; and
3. To use a mixture of the first two.

The first option is the usual approach in sequential implementations, however the issue here is more critical since the communication overhead is not under control and could lead to inefficiencies. The second option is difficult, since everything has to be taken care of, and there is the risk (somehow paradoxical) that the difficulty of this task will lead to a slow and imperfect implementation. The best solution appears to be the third one which is a compromise between the first two
options. The third approach has been used and in this scheme sets are divided in two classes:

(i) Those generated as intermediate results of a computation;
(ii) 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.

\[ \text{doubleIncrSet}([X.]) \text{ contains } \{\text{incr}(\text{incr}(X))\} \]
\[ \text{incr}(X) = X + 1 \]

Figure 4 illustrates the lines of the SAM code generated for this assertion that executes the increment of \(X\) in parallel on all the processors on which the set is stored.\(^2\) 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 only needed 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 i while the result of \([7]\) to class ii. The choice of this design is to store a set of class i in the same cells as the original set and to store a set of class ii in a new cluster of cells. Since the abstract design of the SAM associates its own cell to each set element, this kind of implementation amounts to explicitly handling the virtualization for sets of class i and leaving this task to the underlying system for sets of class ii.

Such a choice yields an efficient and fast way of generating intermediate sets, which are only needed locally and not globally. They do not, therefore, need to be burdened by communications. Furthermore, this choice keeps a simple structure for global sets: their elements maintain the one to one correspondence with an AM cell, possibly a virtual one, aiding the task of bookkeeping the system, identifying global sets, dereferencing their elements and so on.

4. Structure of the active memory

The implementation of the AM on the connection machine (CM-AM) is built on two major entities:

1. The active memory itself, which contains the sets, elements in its cells;
2. The CM-data-array, which is used to identify the various sets in the active memory.

\(^2\)This is just a scheme of the real code; not all the SAM instructions needed are placed and the arguments are left out in order to keep the presentation simpler. The details that are omitted are presented in the section about mapping.
Figure 5 illustrates the structure of the AM: it is the usual array of cells. In this implementation a register of each cell, the tag register, is used to identify a permanent set. Other sets’ elements can be stored in this cell, but they belong to temporary sets. The element of the permanent set is stored in the first register of the cell. A permanent set in the AM is therefore represented as a collection containing what is in the first registers of the cells that have the same value in the tag register. A permanent set is therefore identified by just a tag.

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

An example of this design is shown in Fig. 6. There are five permanent sets and one temporary set. The permanent sets are identified by tags 1, 2, 3, 4 and 5 and are:

\[
\{a, b, c, d, e, f, g, h, i, l, m, n, o, p, q, r\} \\
\{"otute", "tite", "tibi"\} \\
\{z(x), u(w)\} \\
\{1, 2, -2\} \\
\{1, 4\}
\]

There is an intermediate set, \{1, 4\}, identified by tag 4, with elements placed in the second register of the cells. This intermediate set can be thought of as the intermediate result of the computation of \text{foo}(\{1, 2, -2\}), where \text{foo} is defined in Fig. 7. The argument set can be that identified by tag 4 and the resulting set that identified by tag 5. In the process of computing the result, this intermediate set is created: it contains the square of the elements of the argument set.
Fig. 6. Active memory at work.

\[
id(X) = X.
\]
\[
(X) = X \times X.
\]
\[
\text{foo} (\{X | \_\}) \text{ contains } \{\text{id}(\text{square}(X))\}.
\]

Fig. 7. Permanent and intermediate sets.

Note that here there are two cells containing the element 4. The reason for this is that the duplicate check is performed only when the permanent set is created and not on the intermediate sets. Such design is due to the fact that the extra computations performed are in data parallel with those required, therefore there is no extra time overhead. Furthermore, there is no extra space overhead, since the cell with the duplicate would not be used anyway. It is therefore wise to postpone the duplicate check till a standard set is created; a complete description of the problem can be found in Reference 15.

5. Addressing the sets with the CM-data-array

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 (Fig. 8) and in the present version of the CM-SAM each entry of it identifies a set. Therefore any references to sets in the front end are references to entries in this table which provides all the information that are needed to completely identify a set in the active memory. Each entry of the CM-data-array contains the following fields:

1. \textit{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);
(2) *Tag*, which is the tag used in the tag register of the cells of the set in the active memory to identify which cells belong to a set;

(3) *Register number*, which determines in which register of the cells the elements of an intermediate set are stored, it is 0 for permanent sets.

Figure 9 presents a possible structure of the CM-data-array for the situation identified by Fig. 7. The front end identifies with entry 3 and 5 of the CM-data-array the permanent sets with tags 4 and 5, respectively, whereas the temporary set located at the second registers of cells with tag 4 is addressed by entry 4.

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 possible to have a higher degree of modularization. Then there is a higher degree of flexibility of the whole structure since the code never directly addresses objects in the hypercube. Furthermore, there may be different entries in the table pointing to the same set in the AM; this can be achieved when ad hoc abstract analysers, which are now under development [14], determine that two sets are identical; it is then possible to 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 use of the CM-data-array which includes all the previous ones: it is the way in which the mapping operations are performed.

6. Parallel execution on the active memory

Mapping is perhaps the operation that most exploits the data parallel structure of the active memory. The mechanism which is used to perform a mapping is therefore explained. This explanation is performed referring to the doubleIncrSet clause, which is

\[ \text{doubleIncrSet}([x], \cdot) \text{ contains } \{ \text{incr}(\text{incr}(x)) \} \]
\[ \text{incr}(x) = x + 1 \]

The complete SAM code for doubleIncrSet is given in Fig. 10. Figures 11–13 are snapshots of the active memory, the CM-data-array and the registers after the execution of line [12] of doubleIncrSet upon the call doubleIncrSet\{1,2,3\}. Such a call produces the following execution: Line [2] allocates on the stack the frame for the clause; lines [3] and [4] are used to dereference the argument of the clause and to prepare the space for the result; line [5] starts the mapping. From now on the argument set, identified up to now by \(Y_1\), is referred to element by element, not as a global entity. For this reason a new entry (entry 2) is inserted in the CM-data-array with the tag DDT, meaning distributed data. Any reference to such entries are ‘distributed

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

Fig. 10. SAM code for doubleIncrSet.
Fig. 11. Active memory for doubleIncrSet.

references to all of its elements in the active memory. However, since this entry refers exactly to the same set as Y1 (entry 0), the tag is still 1 and the register number is 1. The permanent register Y4 contains a reference to this entry. Therefore at this point there are two registers, Y1 and Y4, referencing the same set under two different points of view. Register Y3 is kept in the map over for backward compatibility with other SAM implementations and for forward compatibility with other potential versions partitioning the set in clusters; here it is meaningless.

Lines [6] and [7] prepare the argument and the result for the call of line [8]. In the context of a mapping, a new DDT is needed to store the result of incr/2 of line [8]. Entry 3 in the CM-data-array identifies this set which is defined by tag 1 and register number 2; Y5 refers to such an entry. Lines [9] to [11] are analogous to lines [6] to [8] referring to entry 4 in the CM-data-array, to the permanent variable Y6 and to the third registers of cell with tag 1. Up to now no interprocessor communication has occurred in the active memory: everything has been handled locally.

Line [12] performs the actual copy of the set in its final position: the duplicate check is performed

Fig. 12. CM-data-array for doubleIncrSet.
Declarative languages using CM2 active memory

\[
\begin{array}{c|c|c}
\text{Y6} & \text{DA} & 4 \\
\text{Y5} & \text{DA} & 3 \\
\text{Y4} & \text{DA} & 2 \\
\text{Y3} & \text{DA} & 0 \\
\text{Y2} & \text{DA} & 1 \\
\text{Y1} & \text{DA} & 0 \\
\end{array}
\]

Fig. 13. Registers for doubleIncrSet.

\[
\begin{align*}
\text{sumSet}(\emptyset) &= 0. \\
\text{sumSet}(\{X\mid T\}) &= X + \text{sumSet}(T). \\
\text{nugae}\{\text{Element}\mid\text{Remainder}\} &\text{ contains } \{\text{Element}\ast\text{SumSet(Remainder)}\}.
\end{align*}
\]

Fig. 14. SEL code for nugae.

at this stage. Line 13 provides the cleaning up of the CM-data-array and of the active-memory after the end of the computation and line 14 deallocates the stack.

At the end of this sample execution, the crucial role that the CM-data-array plays clearly has an efficient, as well as clean, porting of the SAM on the CM2.

Note that it is possible to have intermediate sets which are not ‘distributed data’, i.e. that need to be handled as a global entity like any permanent set. An example of this situation is presented in Fig. 14. In the clause defining nugae, Remainder must be handled not as a distributed data but as a global set. For such a situation, another tag is available in the CM-data-array: TMP, indicating that a set is an intermediate one, but it must be handled as a global entity [16].

7. The parallel environment

The connection machine is a SIMD machine, therefore only one instruction can be executed at a time on all the processors. This peculiarity needs some special care, since there are data parallel flows requiring 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 must be performed on the other. A mechanism, called parallel environment, has been devised in order to take care of it. Figure 15 presents a situation where such a parallel environment is used. The SAM code for the clauses oneProducts and allProducts is in Fig. 16. A call to allProducts({1,2,3},\{4,5,6\}) may result in the following flow:

Lines [1] to [5] of allProducts are executed, resulting in the following register assignments:
oneProducts(X, {Y | _}) contains {X*Y}.
allProducts({X | _}, S) contains oneProducts(X, S).

Fig. 15. Need of parallel environments.

Y1 ← \{1, 2, 3\}, Y2 ← \{4, 5, 6\}, Y3 ← \(\text{result}\). Then the mapping starts (line [6] and Y5 points to \{1, 2, 3\}) as distributed data. Lines [7], [8] and [9] prepare the environment for the call of oneProduct. Note that by now everything is in data parallel on the set \{1, 2, 3\}. Line [10] executes the call saving the current environment (status and data registers and so on) on the stack. Then, lines [1] to [5] of oneProducts are executed. The register status is now: Y1 ← \{1, 2, 3\} as distributed data, Y2 ← \{4, 5, 6\}, Y3 ← \(\text{result}\). Line [6] starts a new mapping; this mapping is on the set \{4, 5, 6\}. But at this point there is already a data parallel execution in action: that on set \{1, 2, 3\}. However, because the CM2 is a SIMD machine it is not possible to have two mappings acting in parallel because this would result in breaking the restriction of having just one instruction executed at a time.

It is therefore necessary to sequentialize the data parallelism on the first set starting the data parallel execution on the second set. The end_map_over of line [12] will restart the data parallelism on \{1, 2, 3\}. Consequently, it is necessary to provide a mechanism for 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 parallel 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 analysed 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 order to start the sequentialization (for instance 1) and it is marked as ‘already analysed’ in the environment. Any reference to Y1 is now a reference to 1. Therefore line [7] puts a 1 in register A1. Then the standard flow is followed in lines from [8] to [11]. Line [12] is the end of

---

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

\[oneProducts/3:\]
\[allocate\]
\[get_variable A1 Y1\]
\[get_set A2 Y2\]
\[get_variable A3 Y3\]
\[map_over Y2 Y4 Y5\]
\[begin: put_value A1 Y5\]
\[put_value A2 Y2\]
\[put_variable A3 Y6\]
\[call \times/3\]
\[insert Y3 Y6\]
\[end_map_over Y4 Y5 begin\]
\[end: deallocate\]
\[proceed\]

---

Fig. 16. SAM Code for oneProducts and allProducts.
the mapping. At this point the parallel environment is popped from the stack and it is analysed to see if the processing has been completed. This is not the case here, since elements 2 and 3 of \{1, 2, 3\} have not yet been processed. A new element is then picked out from the parallel environment, for instance 3, and the environment is saved back on the stack. Any reference to Y1 is now a reference to the number 3. Again lines [8] to [11] are executed and line [12] is reached. The parallel environment is again popped from the 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] to [11] are executed and line [12] is reached. Now all the set \{1, 2, 3\} has been processed therefore the environment can be popped from the stack, the original data parallelism on \{1, 2, 3\} can be resumed and lines [13] and [14] are executed. The execution then goes back to allProducts which is completed in the normal way.

The need for a parallel environment is present not only when there are nested calls to clauses inside mappings but any time there is a nested mapping. Therefore it is also necessary in situations like that of cartProduct which computed the Cartesian product of two sets.

\[
cartProduct(\{X\}, \{Y\}) \text{ contains } \{\text{pair}(X, Y)\}
\]

8. Experimental results

At the time of writing, the first benchmarks have been performed. It is interesting to compare them with those of SICStus PROLOG, one of the best PROLOG implementations, based on the WAM. The SICStus tests, as well as sequential SAM tests, are executed on a RISC Sun 4 running SunOS 4.0.3. The parallel SAM tests run on a connection machine 2 with 8K processors and a Sun 4 front end. Two kinds of results are reported for the CM-SAM: one taking into account the time for checking for duplicates and one not considering it. The reason for such a choice is that the structure of the CM-SAM often makes the process of detecting easy when checking for duplicates can be avoided.

The SEL version of the tests is:

\[
\begin{align*}
\text{incrMap}(\{X\}) & \text{ contains } \{X + 1\} \\
\text{evenFilter}(\{X\}) & \text{ contains if } (\text{even}(X)) \{X\} \text{ else } \\
\text{sumFold}(\{\}) & = 0 \\
\text{sumFold}(\{X \mid T\}) & = X + \text{sumFold}(T)
\end{align*}
\]

While the PROLOG version is:

\[
\begin{align*}
\text{incrMap}([], []) \\
\text{incrMap}(\{X \mid T\}, \{Y \mid W\}) & :- \text{Y is } X + 1, \text{incrMap}(T, W) \\
\text{evenFilter}(\{X \mid T\}, \{X \mid W\}) & :- \text{even}(X),!, \text{evenFilter}(T, W) \\
\text{evenFilter}(\{\} \mid T, W) & :- \text{evenFilter}(T, W) \\
\text{sumFold}([], 0) \\
\text{sumFold}(\{X \mid T\}, W) & :- \text{sumFold}(T, Z), \text{W is } X + Z
\end{align*}
\]

Table 1 summarizes the results.
Table 1. Summary of comparison.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>SICStus</th>
<th>SAM</th>
<th>CM-SAM With duplicate checking</th>
<th>CM-SAM Without duplicate checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>incrMap</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>7</td>
<td>0.22</td>
</tr>
<tr>
<td>incrMap</td>
<td>100</td>
<td>180</td>
<td>180</td>
<td>50</td>
<td>0.22</td>
</tr>
<tr>
<td>incrMap</td>
<td>1000</td>
<td>1620</td>
<td>3300</td>
<td>600</td>
<td>0.22</td>
</tr>
<tr>
<td>incrMap</td>
<td>10000</td>
<td>15000</td>
<td>405017</td>
<td>5800</td>
<td>0.22</td>
</tr>
<tr>
<td>evenFilter</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>8</td>
<td>0.27</td>
</tr>
<tr>
<td>evenFilter</td>
<td>100</td>
<td>169</td>
<td>175</td>
<td>56</td>
<td>0.27</td>
</tr>
<tr>
<td>evenFilter</td>
<td>1000</td>
<td>1740</td>
<td>3280</td>
<td>620</td>
<td>0.27</td>
</tr>
<tr>
<td>evenFilter</td>
<td>10000</td>
<td>17300</td>
<td>406034</td>
<td>6</td>
<td>0.27</td>
</tr>
<tr>
<td>sumFold</td>
<td>10</td>
<td>21</td>
<td>14</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>sumFold</td>
<td>100</td>
<td>131</td>
<td>40</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>sumFold</td>
<td>1000</td>
<td>1010</td>
<td>450</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>sumFold</td>
<td>10000</td>
<td>11360</td>
<td>3900</td>
<td>1.22</td>
<td>1.22</td>
</tr>
</tbody>
</table>

The results are encouraging. The SAM behaves pretty well in comparison with SICStus, which is one of the best and fastest PROLOG compilers currently available: the SAM challenges SICStus in incrSet and evenFilter for small and medium sized sets (up to 1000 elements) and outperforms it in sumFold for any size. The problem connected with large sized sets is that of checking for duplicates. Abstract analysers are under development to determine when this operation is necessary [7, 15]. These performances were obtained without any ad hoc optimization, whereas SICStus is implemented with many peephole optimizations: there is also much room for improvement in the sequential case.

On the connection machine, checking for duplicates carries a linear cost. This explains the linear increase of time for the execution of incrSet and evenFilter, while no duplicate check is required for sumFold. However, quite often duplicate checks can be avoided, or at least deferred till the end of the computation. Consider the definition of severalMaps:

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

There is no need to perform the duplicate check operation until after the call of f [15]. The time required without checking for duplicates is also presented for these reasons. Note that it is almost constant for any assertion. This means that the structure of this machine exploits 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.

9. Conclusions

The core of the data parallel implementation of SEL on the CM2, the active memory, has been described. The active memory is distributed on the hypercube of processors of the CM2 taking...
advantage of the virtualizing facilities offered by the CM2. The sets are handled using a global
table, the CM-data-array, which is stored in the front end. To take care of the fact that the
CM2 is a SIMD machine a special mechanism, called a parallel environment, has been devised.
The early results of this approach are promising. A full implementation of the SAM is expected
soon.

Several issues are still open. Abstract analysers and garbage collectors are under study and
development. A different way of mapping the active memory on the hypercube by means of hash
tables is almost complete. Network optimizations and compilation enhancements are also hot
points.

Acknowledgements

Most of the ideas presented in this paper have been discussed with B. Jayaraman. The authors
thank researchers of DII/Parma for their support in the use of the connection machine.

References

1. A. Calderwood and P. Szeredi. Scheduling Or-parallelism in Aurora: The Manchester Scheduler, in
G. Levi and M. Martelli (editors) Proceedings of the Sixth International Conference on Logic
2. E. Tick. A Performance Comparison of AND- and OR-Parallel Logic Programming Architectures, in
G. Levi and M. Martelli (editors) Proceedings of the Sixth International Conference on Logic
M. Martelli (editors) Proceedings of the Sixth International Conference on Logic Programming, Lisbon,
4. G. Succi and G. A. Marino. Exploiting the Data Parallelism of Subset Equational Languages, in
5. G. Succi. Exploiting Implicit Parallelism of Logic Languages with the SAM, in Proceedings of the 1992
7. B. Jayaraman and A. Nair. Subset-Logic Programming: Application and Implementation, in 5th
10. G. Succi and G. A. Marino. The Design of an Abstract Machine for Subset Equational Languages, in
Proceedings of the 1992 European Workshop on Parallel Computing, Barcelona, Spain, March 1992 (IOS
Press).