MULTIPROCESSING IMPLEMENTATION OF A HIGH-LEVEL MACHINE LANGUAGE

LUCA CARDELLI^, GIANFRANCO PRINI^^, MARCO VANNESCHI

^ DEPARTMENT OF ARTIFICIAL INTELLIGENCE-UNIVERSITY OF EDINBURGH
HOPE PARK SQUARE - MEADOW LANE - EDINBURGH EH8 9NW - SCOTLAND

^^ ISTITUTO DI SCIENZE DELL'INFORMAZIONE- UNIVERSITÀ DI PISA -
CORSO ITALIA 40 - I-56100 PISA - ITALY

We present a high-level machine language called SMOM. It is a conventional zero-address machine language containing several high-level functions for defining and manipulating data types. Though SMOM does not allow parallel control regimes to be explicitly programmed, it exibits a considerable amount of implicit parallelism, since it is an applicative language and arguments of functions may be concurrently evaluated. This observation is the guideline for the definition of a new simple scheduling discipline that allows SMOM programs to be executed by several co-operating processors. The proposed technique consists in enqueuing a function on all processors which are still performing some subordinate computation relative to that function (i.e. evaluation of some of its arguments). Only one of this processors is then enabled to actually compute the function, namely the one which terminates the last subordinate computation. A multiprocessor based on the proposed technique can be actually realized with existing hardware. The speed-up in the execution of SMOM programs is expected to be one order of magnitude.

1. INTRODUCTION

Several programming languages, such as ALGOLW, SIMULA 67, ALGOL 68, PASCAL, EL1 and CLU, have been designed to allow for data extensibility, that is the ability of declaring (vs. explicitly programming) data types. Since most computers are not very well suited to support extensible languages (10), designers are almost inevitably induced to restrict the mechanisms for declaring data types, mostly to the purpose of minimizing run-time type-checking. The problem of designing and efficiently implementing a truly general extensible language would be greatly simplified by using fast and sophisticated memory managers.

In (3) a machine, called SMOM, is presented which allows data types to be specified at the machine language level, in a way similar to (8,11). SMOM may be considered as a powerful memory manager which operates under the control of a zero-address, stack-oriented machine language. One of its explicit design goals has been to allow for efficient implementations of general extensible programming languages (for a somewhat elaborate discussion of this issue see (3)). Here we point out how such a machine language may be supported by a multiprocessor architecture which significantly speeds up the execution of SMOM programs by concurrently evaluating arguments of functions and by performing most type-checking at run-time, but in parallel with other significant computations.

Section 2 contains a short but self-contained description of SMOM. In Section 3 the characteristics of parallel processing of SMOM code, including a unique interprocess communication discipline, are presented. They lead to the general ideas underlying SMOM architecture. In Section 4 a multiprocessor organization is suggested which implements the ideas of Section 3. Some concluding remarks are contained in Section 5. A detailed description of some relevant SMOM ma-

Pervenuto alla redazione: Novembre 1977

chine instructions is contained in the Appendix.

2. SMOM: A SHORT DESCRIPTION

2.1. Data types

All data are manipulated through data descriptors (dd's). A dd is a typed pointer, i.e. a pair (T,A), where T is the type of the described datum, and A is its memory address. Both types and addresses are manipulated by the hardware only: the user is not enabled to access them. This implies that memory management is completely transparent to the user. Data may be either basic (i.e. with no inner structure) or structured (i.e. arrays and records of dd's). Both basic and structured data types may be either built-in or user-defined.

A data type is introduced by defining a class. A class is a structured data type that records all information relative to a specified data type. It contains a template which defines the inner structure of all data of that type (the template being empty for basic data types), and an unspecified number of locations, called registers, whose main purpose is to record (dd's for) the definitions of the functions associated with that class (see Section 2.3 and 2.4). Thus a class may be seen as a data type with an associated behaviour.

As an example, Fig. 1 shows the declaration of the class of COMPLEX numbers, which are records with two fields, called REALPART and IMAGPART, containing two REAL numbers.

```
(CLASS COMPLEX
   (TEMPLATE (REALPART REAL)
             (IMAGPART REAL)
   (REGISTER ADD
             <code for ADDing two
             COMPLEX numbers>
       .
       .
       .
   (REGISTER FOO
             <code for FOOing two
             (three?) COMPLEX
             numbers>))
```

Fig. 1. The definition of the class of COMPLEX numbers.

Evaluation of class definitions is explained in Section 2.3.

2.2. Stacks

In order to process a datum, a dd for it must be present in the argument stack (A-stack), whose building blocks are the A-cells. Though the A-cells contain dd's, for short they are also said to contain data. The A-cell at the top of the A-stack is named A-top. The datum contained in the A-top is the top-datum, its type is the top-type.

Continuation points of programs (see below) are held in the control stack (C-stack), which is made of C-cells. The C-cell at the top of the C-stack is named C-top.

2.3. Microprograms

Two special built-in data types are present in SMOM, namely programs and microprograms. Microprograms implement the basic computational functions of SMOM (i.e. they are supposed to be executable by the hardware). Microprograms may be grouped into the following categories.

a) A-managers and C-managers are used for the data control (i.e. for the management of the A-stack) and for the sequence control (i.e. for the management of the C-stack) respectively. The main A- and C-managers are described in the Appendix. Their descriptors are stored in the homonymous registers of the special class ANY (see below).
b) Operations are used for standard computations. Each of them has an associated arity (i.e. number of arguments). The execution of an n-ary operation (i.e. an operation whose arity is n) causes n arguments to be popped off the A-stack and a value to be pushed onto the A-stack itself.

SMOM contains several built-in microprograms (i.e. all A- and C-managers and microprograms which specify the behaviour of built-in data types). Many others (mostly operations) are automatically synthesized when a class definition is evaluated, and dd's for them are stored in the appropriate registers. For instance, referring to the example of Fig. 1, the following functions are generated.

a) A constructor operation for building new COMPLEX numbers out of two REAL numbers. Its entry point is stored in the register COMPLEX of the class ANY.

b) Two selector operations for accessing the components (i.e. the REALPART and the IMAGPART of a COMPLEX number). Their entry points are stored in two registers, named REALPART and IMAGPART, of the class COMPLEX. They may operate either in load mode (in which case they are unary and are used to retrieve the components of COMPLEX numbers) or in store mode (in which case they are binary and are used to update the components of COMPLEX numbers: the first argument replaces the appropriate component of the second argument). The default mode is the load mode. The store mode is used when a selector is invoked via the special operation UPDATE (see Section 2.4).

c) Several utility operations for reading, printing, editing, etc. COMPLEX numbers according to a standard representation. Their entry points are stored in the registers READ, PRINT, EDIT, etc. of the class COMPLEX.

The mechanism which allows microprograms to be retrieved and executed is explained in Section 2.4.

2.4. Programs

Programs are linear sequences of functions, each one being made of an operation code and some (possibly zero) operands. Both operation codes and operands are represented as bytes: thus a program is represented as a linear sequence of bytes. Each byte is interpreted either as an operation code or as an operand according to its position, as it usually happens in a standard byte-oriented computer.

Programs are interpreted by a fetch-decode-execute loop which performs the following operations.

A byte is fetched, according to the address contained in the C-top (fetching always increments the contents of the C-top), and it is interpreted as an operation code. This means that a register having the same internal name as the fetched byte is searched in the special class ANY. If such a register is found, the datum described by it is used as a function according to the following rules.

a) If the datum is a microprogram, it is directly executed by the hardware. It may fetch some bytes of the calling program and use them as operands: this is the case of all the functions described in the Appendix (other than RETURN) and the function UPDATE (which has the name of a selector as its only operand).

b) If the datum is a program, a dd for it (i.e. a pointer to its first instruction) is pushed onto the C-stack and execution goes on with the new program counter. Note that recursive calls are performed naturally, and recursion may be implemented straightforwardly, provided that the A-stack (which is used for passing parameters) is properly synchronized with the C-stack. This means that, at the end of a recursive call, a program should pop its parameters off the A-stack and replace them with a value.

c) Otherwise the datum is pushed onto the A-stack and the standard function APPLY is called. By the type-driven call mechanism (see below) the function APPLY is searched in the class which contains the datum itself. Thus, each datum may be used as a function, provided that a function APPLY (i.e. an interpreter) is defined for it.

If the class ANY does not contain a register with the same internal name as the fetched byte, such a register is searched in the class of the top-datum. If such a register is found, processing goes on as in the previous case, otherwise an error is generated. The call mechanism just described is named type-driven, since the top-type "drives" the access to the code of the called function.

An extensive discussion of the relations between the notion of class plus type-driven function invocation and more familiar concepts like "user defined data types" (in the sense of PASCAL) or "abstract data types" is outside the scope of this paper, and may be found in (2). Here we limit ourselves to observing that SMOM classes are more similar to "abstract data types" than to "user defined types".

A sample program which performs the addition of two COMPLEX numbers is shown in Fig. 2. The evolution of the A- and C-stacks during the execution of such program is shown in Fig. 3. Note that, although the program is written in a "pretty format", its memory representation is simply a sequence of bytes, the first byte being the internal name of "GET", the second one being the binary code of "1", etc.

```
(PROG
   (GET 1) (REALPART)
   (GET 1) (REALPART) (ADD)
   (GET 2) (IMAGPART)
   (GET 2) (IMAGPART) (ADD) (COMPLEX)
   (SQUEEZE 2)
   (RETURN))
```

Fig. 2. A program for adding two COMPLEX numbers.

```
C-stack = (PROG (GET 1) ... )
A-stack = c+id
          a+ib

C-stack = ((REALPART) (GET 1) ... )
A-stack = a+ib
          c+id
          a+ib

C-stack = ((GET 1) (REALPART) ... )
A-stack = a
          c+id
          a+ib

C-stack = ((ADD) (GET 2) ... )
A-stack = c
          a
          c+id
          a+ib

C-stack = ((GET 2) (IMAGPART) ... )
A-stack = <value of a+c>
          c+id
          a+ib

C-stack = ((COMPLEX) ... )
A-stack = <value of b+d>
          <value of a+c>
          c+id
          a+ib

C-stack = ((SQUEEZE 2) ... )
A-stack = (a+c)+i(b+d)
          c+id
          a+ib

C-stack = ((RETURN))
A-stack = (a+c)+i(b+d)
```

Fig. 3. The evolution of SMOM stacks while executing the program of Fig. 2. The irrelevant parts of the stacks (i.e. deeper programs and data) have not been displayed for simplicity.

3. SMOM ARCHITECTURE AND PARALLELISM

3.1. General discussion

In Section 2 SMOM functions have been supposed to be executed in a strictly sequential order, the order being specified by contiguity and jump operations (program calls are to be considered as jump operations). However, the features of SMOM make it possible to exploit the inherent parallelism of the algorithms in a natural way.

As it happens with high-level expression languages (like LISP), a SMOM computation consists of a number of function calls, where function bodies are expressions. Thus the computation graph consists of a series of directed acyclic graphs (i.e. partial orders). This is not true of imperative languages (in particular conventional assembler languages) where this property holds locally: essentially it is confined to "small" arithmetical and logical expressions. Moreover, since SMOM functions are more powerful than conventional assembler statements, exploiting the parallelism among them leads to much faster and efficient computations and, as we shall see, to an easily recognizable modular separation of firmware-controlled system tasks.

We now intend to show how to detect, order and execute in parallel the various parts into which a program may be decomposed. Of course, determinacy and efficiency impose suitable synchronization requirements: they will be defined in a unique way.

In what follows, the reader is recommended not to take care of the type-driven call mechanism of SMOM. Also the existence of functions (such as PUT and UPDATE) that perform side-effects is to be ignored by now. These features will be considered later.

Referring to Fig. 2, it is evident that the computation graph of that program is the tree shown in Fig. 4. Each leaf represents a function which starts an independent computation. Each nonterminal node with K sons represents the execution of a function with K subordinate computations.

Partially ordered computations may be reproduced in a multiprocessor environment by means of several synchronization techniques, all of them being implementations of Dijkstra's P and V primitives (9) or, equivalently, of the Conway's FORK and JOIN instructions (5). However, the efficiency of such techniques is fully exploited in the classical multitasking environment, where tasks are relatively large in size, so that a considerable processor-switching overhead is tolerable (15). Improvements can be devised when the tasks may be of any size (see for instance (6,7,14)), but all these techniques have the same characteristics: every task is dispatched to one processor only as soon as it is ready for execution. This fact prevents from employing other forms of parallelism, such as prefetch or look-ahead: we wish to employ such forms indeed, as we are concerned with the efficient execution of (high-level) machine code. Moreover, we need a synchronization strategy that agrees with the data control (i.e. A-stack management) strategy. Our solution is characterized by the fact that the fetched functions are dispatched to all the processors which are potential candidates to their execution: function execution will be actually controlled by a JOIN-type mechanism and by a data-driven control mechanism. More precisely, functions represented by different leaves are enqueued on different processors for execution. Functions represented by a nonterminal node are enqueued on all the processors which are still involved in some subordinate computation of the node itself. The number of such processors is recorded by a counter associated with that node. When a processor takes a function from its queue, it decrements the counter and executes the function only if the counter is zero. This JOIN-type mechanism prevents a function from being executed unless all of its subordinate computations have been completed.

The enqueuing strategy just explained may be realized by a specialized processor, the Control Processor (CP), which operates in parallel with the general processors (simply called processors) $P_1, \ldots, P_n$ and masters the whole system. The CP operates by prefetching the functions, whenever possible, and by dispatching them to a number of processors and priming the synchronization mechanism.

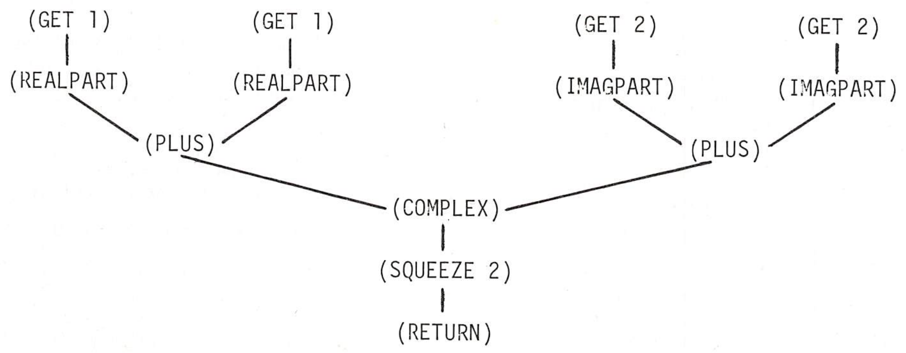


Fig. 4. The computation graph of the program of Fig. 2.

The architecture of a multiprocessor SMOM interpreter is therefore of the hierarchical type. The overall organization is shown in Fig. 5. As we shall see, it is convenient that the A-stack is managed by an independent specialized processor, i.e. the A-stack processor (AP), tightly interacting with CP and the general processors.

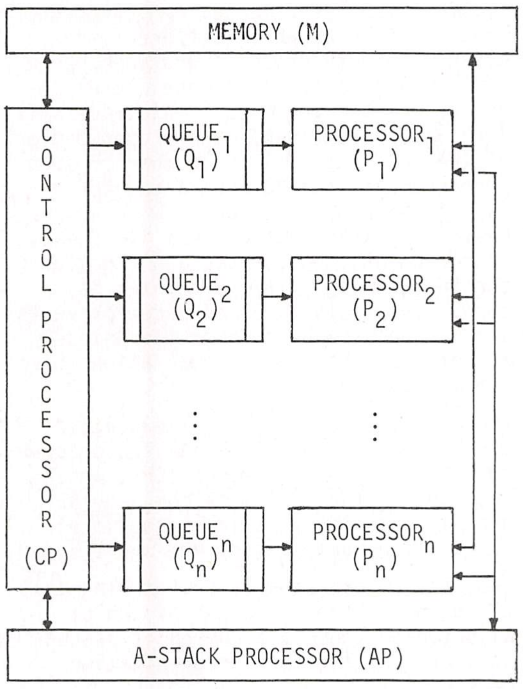


Fig. 5. The architecture of a multiprocessor SMOM interpreter.

In the following subsections several implementation problems arising with such an architecture are examined in some detail.

3.2. The A-stack

Assume that the CP enqueues the first seven functions of the program of Fig. 2 on the three processors P_1, P_2 and P_3, according to the strategy explained in Section 3.1, as shown in Fig. 6. In the figure some A-cells have been labeled in order to single out the A-cells (also called source A-cells) which contain the

```
A-stack = <unused>  A-cell D
          <unused>  A-cell C
            c+id    A-cell B
            a+ib    A-cell A
              .
              .
              .
```

$QUEUE_1$ = GET(A→C)
REALPART(C→C)
ADD(C,D→C)

$QUEUE_2$ = GET(B→D)
REALPART(D→D)
ADD(C,D→C)

$QUEUE_3$ = GET(A→D)

Fig. 6. The state of the system after prefetching seven functions of the program of Fig. 2.

arguments and the A-cell (also called destination A-cell) which is to contain the result of each enqueued function. Note also that the cell B is the A-top when prefetching is started.

Assume that the processors execute the prefetched functions according to the timing of Fig. 7.

P_1 GET(A→C) REALPART(C→C) ADD(C,D→C)
```
o=============o=============o==========
```

P_2 GET(B→D) REALPART(D→D)
```
---o========o=============o-------------
```

P_3 GET(A→D)
```
------o====================o-----------
```

Fig. 7. A possible timing for the execution of the prefetched functions of Fig. 6.

It is evident that the value deposited by P_2 into the A-cell D is overwritten by P_3 before P_1 is enabled to use it for the addition (remember that P_2 cannot perform the addition, since P_1 is still computing the first addendum when P_2 has already completed the computation of the second one).

This example shows that the A-cells cannot be simply overwritten by the processors without possibly generating disastrous side-effects on the A-stack. As mentioned before, semaphores, critical sections and other well known devices for interprocess communication are not needed to share the A-stack correctly among the various processors. A much simpler solution is sketched in Fig. 8. The A-stack is made into a stack of pointers to A-cells, which in turn still contain dd's. The A-stack proper (i.e. the stack of pointers) is

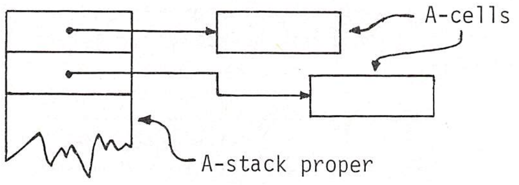


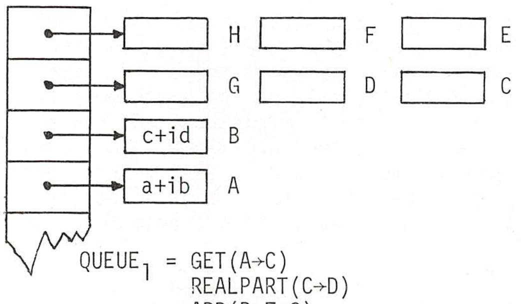
Fig. 8. The structure of the A-stack.

known to the CP only. The A-cells are manipulated by both the CP and the various processors. When a new function (other than a C-manager, which is always executed by the CP, as explained in Section 3.6) is fetched from the memory, the CP enqueues on the appropriate processors the prefetched function and the address of its source and destination A-cells. The former are taken near the top of the A-stack proper, the latter is the address of a newly allocated A-cell, which becomes the new A-top (the A-stack proper is appropriately overwritten). The behaviour of the CP with the program of Fig. 2 is sketched in Fig. 9.

Note that no A-cell is ever overwritten: only the A-stack proper is. However, no information is lost now, since the A-cells containing useful values are still referenced from the processors' queues. When an A-cell becomes unreferenced, it may be reclaimed to free-storage, as explained in Section 3.3.

3.3. A-managers and the A-cell space

In Section 3.2 we have introduced the notion of source and destination A-cells



QUEUE_1 = GET(A→C)
REALPART(C→D)
ADD(D,F→G)

QUEUE_2 = GET(B→E)
REALPART(E→F)
ADD(D,F→G)

QUEUE_3 = GET(A→H)

Fig. 9. The state of the system after prefetching seven functions of the program of Fig. 2.

for a function. These concepts are clear enough as far as operations (see Section 2.3) are concerned. But what about the A-managers? Few problems arise with PUSH: it may be considered as a nullary operation (i.e. an operation with zero arguments) which is completely executed by the CP. Also POP and SQUEEZE generate few problems: they may be considered as (n+1)-ary operations which return their first and last argument respectively, totally ignoring the other n arguments. They may be enqueued and executed according to the strategy explained in Section 3.2.. In the case of both the three A-managers considered by now and all the operations, the following policy may be devised for the management of the A-cell space. Whenever a function is prefetched, a destination A-cell is allocated, as already explained in Section 3.2. Whenever a function is executed, its source A-cells are deallocated, since they cannot be referenced any more.

As far as GET is concerned, almost the same policy may be adopted: a destination A-cell is allocated by the CP, as shown in Fig. 9, but the source A-cell is not deallocated when GET is executed.

Unfortunately, this simple policy does not seem to be applicable to the funtions PUT and UPDATE, unless either the management of the A-cell space is intolerably complicated or the CP is forced to wait until all currently enqueued functions have been completed. The reader is invited to convince himself that any simple solution which does not involve a garbage collector for the A-cell space is actually wrong. Counterexamples are

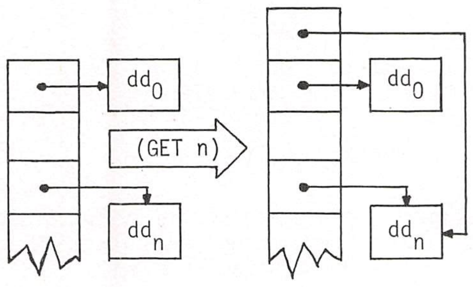


Fig. 10. An implementation of GET.

easily found in which GET and PUT, from the one side, and selectors used in load and store mode, from the other side, do not properly interact, if an appropriate timing is assumed for their execution.

A totally different view may also be adopted: each A-cell is provided with a reference count which records the number of existing pointers to that A-cell. This has a number of pleasant consequences. First of all, the A-managers may now be executed directly by the CP, without allocating extra destination A-cells (see Fig. 10 for an example). Second, PUT may be completely prefetched without unnecessary waitings. Third, no A-cell is to be allocated for unary functions. The main disadvantage of this solution is that the maintenance of reference counts introduces a lot of write operations on A-cells. This may probably cause the performance of the system to be dramatically compromized unless (as already suggested) a special purpose processor, i.e. the AP, is introduced for managing the A-cell space.

3.4. The enqueuing strategy

In Section 3.1 a strategy has been outlined for reproducing computation graphs and dispatching functions to processors. Here we make that strategy effective.

In Fig. 4 each node of the computation graph has been associated with a function. In Section 3.2 each function has been associated with a single and newly allocated A-cell (namely, the destination A-cell). This one-to-one correspondence between nodes and destination A-cells allows us to associate the counters mentioned in Section 3.1 with the A-cells. Such counters have been supposed to be positive integers in Section 3.1. Obviously, an integer counter only contain information about how many (while the CP must know which) processors are co-operating in the computation associated with a node. Hence, an array of boolean indicators is more suitable in this case. Thus, an A-cell may be patterned after Fig. 11.

s_1	s_2	.	s_n	type	pointer

Fig. 11. The pattern of an A-cell with boolean indicators.

At any time, the indicator s_i ($1 \leq i \leq n$) of an A-cell is TRUE if and only if the processor P_i is still involved in some subordinate computation of the node represented by that A-cell. Each indicator of a newly allocated destination A-cell is initially set to the inclusive OR of the corresponding indicators of the source A-cells of the fetched function: in fact these A-cells, which are to contain the arguments of the function, also contain information about the processors which are still computing them. The function is then enqueued (with the address of the source and destination A-cells) on all the processors whose corresponding indicator in the destination A-cell is TRUE. If no such indicator is TRUE, the function may be enqueued on an arbitrary processor, and the corresponding indicator is set to TRUE.

When a processor takes a function from its queue, it sets to FALSE the corresponding indicator in the destination A-cell, and executes the function if and only if all the indicators of this A-cell are FALSE. Otherwise the function is ignored, and the next one is taken from the queue.

3.5. Type-driven calls

Here we explain how the CP may perform the run-time type-checking which is necessary for the implementation of the type-driven calls.

Suppose first that the prefetched function is not a C-manager and its code is found in the special class ANY. In this case the arity of the function is known, and the type of its result may be often predicted (input functions are a major exception). So the CP may update the A-stack properly, and store the predicted type in the destination A-cell before the function is actually computed, thus communicating information to the next (possibly type-driven) call.

If the code of the prefetched function is not found in the special class ANY, the type-driven call mechanism is to be invoked. If the top-type is manifest (i.e. it has been computed by either the CP or some other processor) then the CP operates as in the previous case, using the appropriate class instead of the special class ANY. Otherwise the CP cannot do better than waiting until some processor has computed the top-type.

3.6. C-managers

They can be directly executed by the CP. The execution of TYPEJUMPT and TYPEJUMPF requires the top-type to be manifest, and the CP operates exactly as in the case of type-driven calls (see Section 3.5). The execution of JUMPT and JUMPF requires that the top-datum is manifest, hence the CP must wait until it is.

4. MULTIPROCESSOR ORGANIZATION

4.1. Overall system organization

As shown in Section 3, the overall organization proposed for a SMOM interpreter is a hierarchical multiprocessor, consisting of the following components (see Fig. 12).

a) A Control Processor (CP) at the highest level.
b) A set of identical (general) processors $P_1, P_2, \dots, P_n$ at the lowest level.
c) An A-stack Processor (AP) which manages the processors' and CP's requests for accessing the A-stack, and controls their interactions.
d) A common Memory (M), with m modules $M_1, M_2, \dots, M_m$.

As it often happens in a hierarchical, parallel organization, the existence of a supervisor control level makes both the design process and the achievement of high performance easier, possibly at the expense of less reliability. In our case, the performance is further improved by the use of simple and fast synchronization primitives, by an extensive use of prefetching and by the independent management of the A-stack. A modular organization of the memory, possibly with a suitable separation of information among the modules, is a standard technique for balancing the execution and the memory bandwidths.

Every system unit is microprogrammed and, whenever possible and/or convenient, dynamically microprogrammable, in order to improve system flexibility and extensibility. Moreover, the overall organization is modular in principle and thus well suited to better exploit the microprogramming characteristics. This further improves system efficiency, as each unit has its own microprogramming language and organization, both chosen in such a way as to optimize the performance/cost ratio "on a local basis" (17).

In the following subsections we describe the internal organization of the main parts of the system, according to the general behaviour discussed in Section 3.

4.2. Control Processor

As a primary task, the CP must perform the sequence control of the computation. Therefore, the C-stack must be a private resource of CP. Moreover, it is convenient that the CP contains a private copy of the A-stack, called the Virtual A-stack (VA-stack), in order to reduce the number of conflicts with the processors for accessing the A-stack and to maintain a picture of the contents of the A-cells allocated for prefetched, but not yet executed, functions. It is enough that the VA-cells are composed of only two fields, i.e. the V-type and the V-address. The first is assigned the

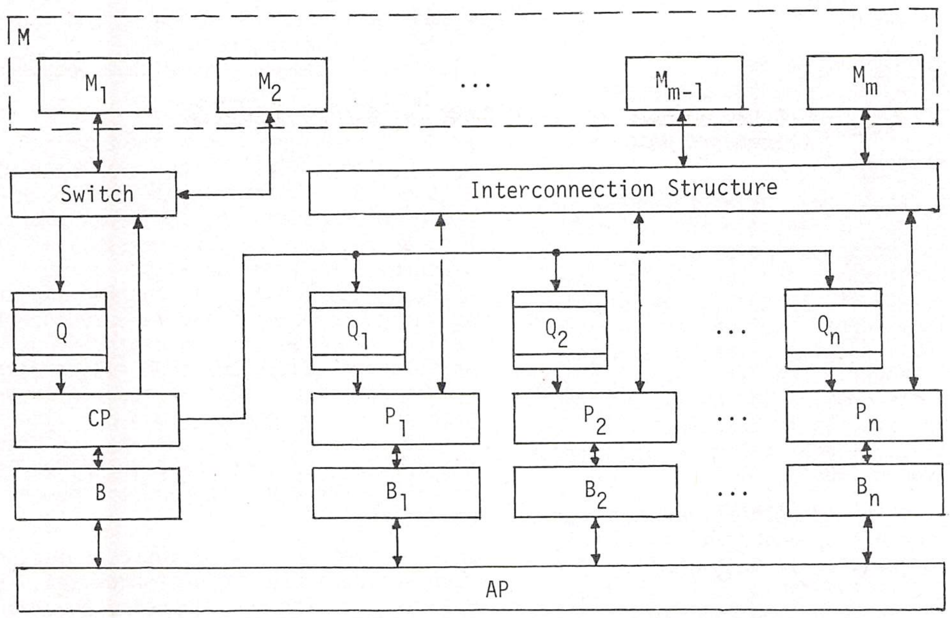


Fig. 12. Overall system organization.

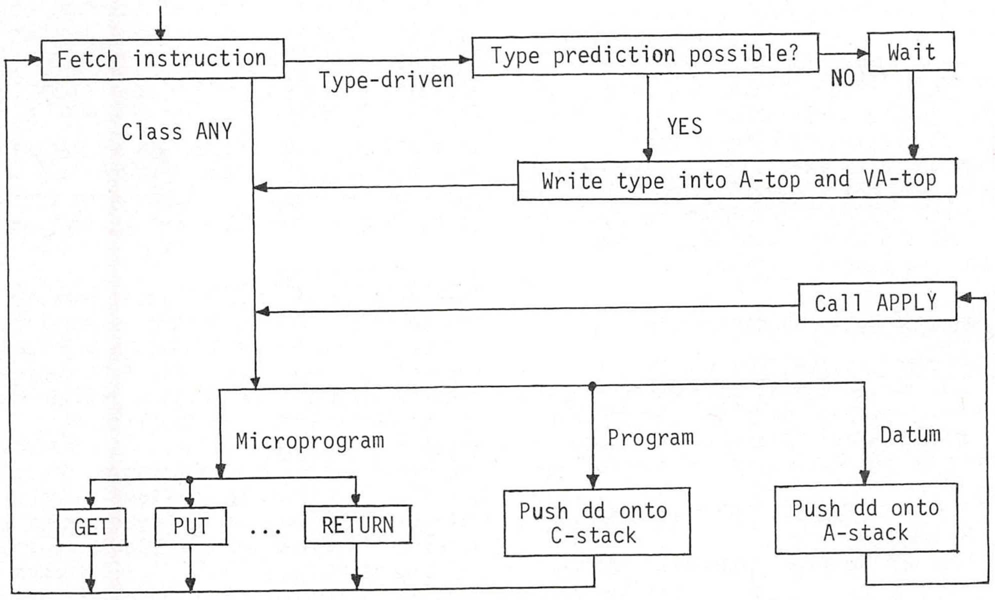


Fig. 13. Scheme of the Control Processor microcode.

type of the result of the currently prefetched function according to well-defined predictions as discussed in Section 3. A special mark in the V-type field signals the wait situation.

The C-stack and the VA-stack are implemented by means of fast memories of relatively small size (typically, 128-256 cells). Two other similar memories are used as branch tables for registers of class ANY and for registers of the other classes. Actually, the second one acts as a cache of the complete table sorted in main memory.

The scheme of the microprogram defining the behaviour of CP is shown in Fig. 13. Remember that the microprograms that affect the control flow are directly executed by the CP. The function APPLY, if present, is an interpreter defined for using a datum as a function: when it is called by the type-driven mechanism, it is searched in the corresponding class of the datum.

As the CP must be as fast as possible for a continuous supply of functions to the processors, it may be realized as a network of bit-slice microprogrammable microprocessors, each one corresponding to a "logical" section of the CP algorithm, all being controlled by a supervisor microprocessor (1,4,14,17).

4.3. General processors

The task of a general processor P_i is that of executing the functions which do not modify the control flow. Its behaviour has been sketched in Section 3 and the scheme of its microcode is shown in Fig. 14.

The internal organization and, correspondingly, the microprogramming type depends not only on the characteristics of the implemented algorithms, but heavily on overall system evaluations too. More precisely, the conflicts for accessing the A-stack and the Memory, together with the degree of parallelism of the computation, impose a saturation to the system performance with respect to both the number of processors and their average speed. Unless we are willing to make the AP organization very sophisticated, at a non-proportional cost, the

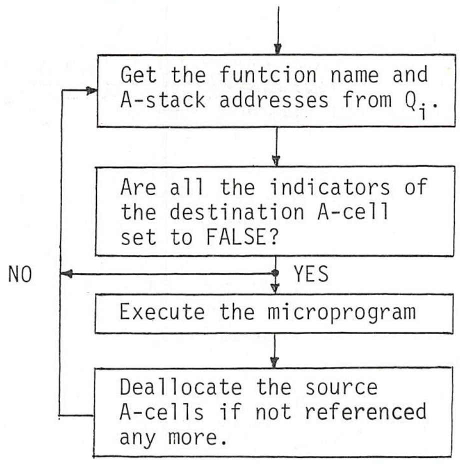

Get the funtcion name and A-stack addresses from Q_i.
Are all the indicators of the destination A-cell set to FALSE?
NO
YES
Execute the microprogram
Deallocate the source A-cells if not referenced any more.

Fig. 14. Scheme of the microcode of a general processor.

threshold speed of the processors seems to be of the same order of magnitude as the typical speed of mini- or microprocessors, as it usually happens with a multiprocessor architecture.

By taking into account the tree form of the computations, a number of 4-8 processors seems the most suitable, unless a multitasking feature is introduced into the SMOM language (by adding special primitives such as FORK and JOIN) and in the system architecture. In this case, the CP must perform the task scheduling. However, some problems arise at the level of the A-stack. It may be interesting to investigate the possibility of employing at the task level the same synchronization strategy proposed in this paper and used at the instruction level.

4.4. A-stack Processor

As discussed in Section 3, the management of the A-stack is a potential bottleneck of the system which may produce serious degradation of the system performance. In fact, if this function is distributed on the CP and the general processors, their behaviour would be severely limited by an excessive overhead. Thus a specialized independent processor, the AP, is needed. The AP is assigned the task of performing functions under the command of the CP and the processors. Examples of such functions are: ALLOCATE and DEALLOCATE A-cells; READ and WRITE A-cells, either with LOCK or UNLOCK feature respectively; conflict resolution and priority updating, queues management, etc.

Commands to the AP are sent by the CP and the processors, and responses are returned from the AP through a set of I/O buffers B, B_1, B_2, ... , B_n (see Fig. 12). The commands specify, together with their code, the relative A-cell address and its configuration if it has to be modified.

The indirect interaction mechanism can be realized, in a simple but effective way, by associating a lock-bit to every A-cell, and by implementing an arbitration algorithm of the general or of the circular type.

A fast memory of hardware registers is employed for implementing the A-stack. A dynamic microprogramming, though desirable, is not essential in this case, for the functions being implemented are easily foreseen during the design phase. Moreover, flexibility can be sacrificed in behalf of speed, owing to the critical task of this unit. The possibility of realizing the AP as a network of asynchronous, co-operating subunits (at the extreme, one for every A-cell) is to be taken into account. However, economy considerations, together with a much less than proportional gain in speed, recommend to follow the proposed solution.

4.5. Memory

In our case, two factors play an important role in devising a cost effective organization of the memory. First, the particular function of the CP, roughly behaving as the Instruction Preparation Unit in a look-ahead computer (15). Second, the possibility of individuating two memory areas, one essentially accessed by the CP, the other accessed by the processors. Therefore, we can assign a subset of modules to the CP and share the remaining modules among the processors. The CP is connected to its modules by a simple switch and a queue (Q in Fig. 12), while the processors are connected through a time-shared bus. This subdivision can be realized in a quite flexible way, provided that the switch and the interconnection structure are implemented in a modular way.

5. CONCLUDING REMARKS

The design goals of the machine language of SMOM were essentially two, namely it had to be particularly well suited to implement fast interpreters for extensible languages, and it had to support the good programming style of defining and using as many data types as logically required by the user's problem. In this case, the types of the components of a datum are almost always known from the definition of the data type which the datum is an instance of. This amounts to saying that the prefetching algorithm seldom falls asleep while waiting for the top-type to become manifest. In other words, a programming style that allows an almost complete compile-time type-checking also allows an equally complete run-time type-checking to be performed by the prefetching algorithm. So, the prefetching algorithm generally waits for the completion of some test having a really unpredictable result, e.g. a test that precedes a JUMPT or a JUMPF.

A hand-made analysis of several SMOM programs (most of them obtained by compiling various LISP functions to SMOM) has shown that about one half of the instructions contained in a SMOM program are A- and C-managers, and that the number of functions which are executed in parallel by the processors (other than CP) is typically varying from four to eight (this result has been recently confirmed by experiments performed with a software simulator of the parallel architecture described in this paper (11)). These numbers perfectly match the number of processors that is conjectured to be optimally supported by the proposed architecture, i.e. the maximum number of processors which does not generate too many conflicts on the A-stack (see also Section 4). This apparently amounts to saying that the maximum speed-up allowed by parallel interpreters of SMOM-like languages is about one order of magnitude. We believe that something better can be achieved, provided that the conflicts for accessing the various memory modules and the A-stack are held to a minimum by a careful design of the algorithms for allocating new data items or new A-cells. Results obtained in implementing extensible high-level languages in paged environments are surprising in this respect. Further improvements in the performance of such systems can be probably achieved only with machine languages based on different concepts (13).

ACKNOWLEDGEMENTS

Dario Citterico and Maurizio Greco participated in early phases of the research that lead to this paper. Their contribution is acknowledged. Gigina Aiello made many suggestions that considerably improved the manuscript.

REFERENCES

(1) A.K. Agrawala & T.G. Rauscher, Foundations of microprogramming: architecture, software and applications (Academic Press, New York, 1976).
(2) L. Aiello, M. Aiello, G. Attardi & G. Prini, Recursive data types in LISP: a case study in type-driven programming, in: B. Robinet (ed.), Programmation (Dunod, Paris, 1976)232-248.
(3) G. Attardi, C. Montangero & G. Prini, A high-level machine for artificial intelligence, in: M. Brady (ed.) Proceedings of the AISB summer conference (Edinburgh University Press, Edinburgh, 1976) 26-37.
(4) T.C. Chen, Microprocessors as building blocks, in: Symposium on distributed computing systems: micros, minis and networks (Pisa, 1975)
(5) M. Conway, A multiprocessor system design, in: AFIPS conference proceedings (1963)136-146.
(6) N. De Francesco, G. Vaglini & M. Vanneschi, Implementation of parallel computation schemata, in: Proceedings of the 2nd Euromicro symposium (Venice, 1976) 157-163.
(7) J.B. Dennis & D.P. Misunas, A preliminary architecture for a basic data-flow processor, in: Proceedings of the 2nd annual symposium on computer architecture (New York, 1975)126-132.
(8) P. Deutsch, A LISP machine with very compact programs, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence (Stanford, 1973) 697-703.
(9) E.W. Dijkstra, Co-operating sequential processes, in: F. Genuys (ed.) Programming languages (Academic Press, New York, 1968)43-112.
(10) E.A. Feustel, On the advantages of tagged architecture, IEEE Transactions on Computers C22,7 (1973)644-656.
(11) B. Giannetti, Valutazione delle prestazioni di un sistema di programmazione integrato, Thesis, Istituto di Scienze dell'Informazione, Pisa (1978).
(12) A. Kay, Personal computing, in: Proceedings of the meeting on 20 years of computer science (Pisa, 1975).
(13) M. Morescalchi, L. Signorini & A. Tomasi, Una architettura parallela per l'interpretazione di linguaggi macchina ad alto livello, in: Atti del congresso annuale dell'AICA (Pisa, 1977).
(14) G. Rietti, U. Tiriticco & M. Vanneschi, Sistemi distribuiti di unità funzionali come complessi di microprocessors, in: Atti del congresso annuale dell'AICA (Pisa, 1977).
(15) H. Stone (ed.), Introduction to Computer Architecture (Science Research Associates, Los Angeles, 1975).
(16) M. Vanneschi, On the microprogrammed implementation of some computer architectures, Euromicro Newsletter 2,2 (1976)14-20.
(17) M. Vanneschi, Microprogrammable processor components and architectures, in: D. Aspinall (ed.), The microprocessor and its applications (Cambridge University Press, Cambridge, 1977).

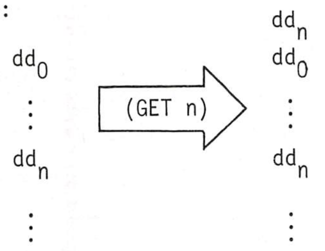
APPENDIX

A.1. A-managers

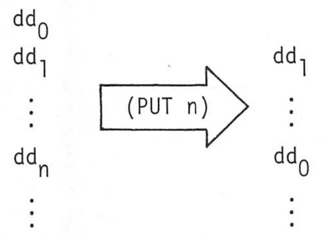
Here n is a non-negative integer which denotes the address of an A-cell relative to the A-top (i.e. 0 is the address of the A-top, 1 is the address of the A-cell immediately below the A-top, etc.). The A-cell whose address is n is also called the n-th A-cell.

Format: (GET n)
Action: copies the n-th A-cell onto the A-stack.

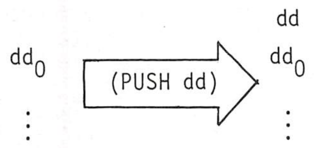
Figure:



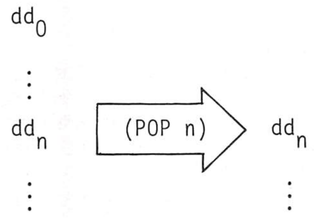
Format: (PUT n)
Action: copies the A-top into the n-th A-cell.
Figure:



Format: (PUSH dd)
Action: copies dd onto the A-stack.
Figure:

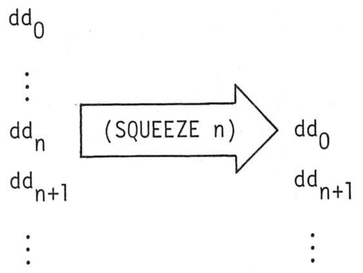


Format: (POP n)
Action: pops n dd's off the A-stack.
Figure:



Format: (SQUEEZE n)
Action: pops n dd's between the A-top and the (n+1)-th A-cell off the A-stack.

Figure:



A.2. C-managers

Here n is a (possibly negative) integer which is to be added to the C-top (such an operation is called a jump).

Format: (JUMP n)
Action: the jump is always performed.

Format: (JUMPF n), (JUMPT n)
Action: the jump is performed if the truth value (basic built-in data type) FALSE (resp. TRUE) is found in the A-top.

Format: (TYPEJUMPF t n), (TYPEJUMPT t n)
Action: the jump is performed if the top-type is not (resp. is) t.

Format: (CALL dd)
Action: pushes dd (which must be a program descriptor) onto the C-stack, thus suspending the program containing the function (CALL dd) and activating the program described by dd.

Format: (RETURN)
Action: pops the C-stack, thus resuming the most recently suspended program.